Static Constructs: Evolution and Impact on Software Quality Aspects

PhD thesis submitted in fulfillment of
the requirements for the doctoral degree
at the Politehnica University of Timişoara
in the field of Computers and Information Technology
by

Eng. Cosmin Marşavina

Thesis committee:
Chair:
University supervisor: Prof. Dr. Habil. Eng. Mihai V. MICEA
Committee members:

Date of thesis defense:

Preface

The thesis was written as part of my PhD at "Politehnica" University of Timişoara. At the beginning I did not fully comprehend what a PhD entailed or the effort required to complete it. With time I managed to put in the work and achieve the goals that were set, thereby allowing me to finalize the studies. This is why I want to express my gratitude to a number of people for aiding me throughout this process.

My special thanks go to Prof. Dr. Habil. Eng. Radu Marinescu with whom I started this journey. He had a big impact on both my professional and personal development. He also gave me the opportunity to start teaching, an experience that I thoroughly enjoy to this day. I would equally like to thank Prof. Dr. Habil. Eng. Mihai V. Micea for coordinating me in the second part of this thesis. Your patience and support have helped me immensely and I cannot thank you enough.

I also want to mention the members of the steering committee: Conf. Dr. Eng. Petru Florin Mihancea, Conf. Dr. Eng. Marius Minea, Sl. Dr. Eng. Dan Cosma, Conf. Dr. Eng. Ciprian Bogdan Chirila, and Conf. Dr. Eng. Codruta Istin. They all provided invaluable feedback on reports, scientific articles, and various other documents that had to be developed as part of the doctorate. I really appreciate that they were always available for discussion, thus helping me make significant progress.

Last but not least, I would like to thank my family and especially my parents, Liviu and Dalila, for their continuous support throughout this period. It has been a long, grueling process and I could not have done it without you.

Timişoara, December 2021 Cosmin Marşavina

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

The chapter provides an introduction to the work that will be presented in the thesis. First, it discusses the problem that is being addressed through this study. Afterwards we introduce the research questions that were formulated and emphasize their importance. The relevance of our work is also explained along with the main contributions that were brought. The last section of the chapter contains the outline for the rest of this thesis.

1.1. Problem statement

Numerous studies have shown that testing is a vital part of the software development life cycle. In [1] Brooks proves that more than half of the effort required for developing complex software systems is spent on testing. The importance of testing is also emphasized in [2] where Sommerville explains the different types of development testing that can be performed and their benefits; the categories mentioned are unit testing, component testing, and system testing. For this study we will focus on unit testing in an object-oriented context as these tests are directly related to specific parts of the source code. The benefits of unit testing are discussed more in depth in [3], which describes how it should happen during each development stage in order to be efficient.

Closely related to the testing process is the testability aspect of software systems. In [4] testability is defined as "the degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met." According to ISO 9126-1 [5] testability is "the capability of the software product to enable modified software to be validated". Other publications (such as [6]) define testability in terms of the effort required for testing. This software quality aspect has proven difficult to quantify. Most of the articles that address software testability assess it during the design and analysis phase, they do not evaluate it based on already implemented code. Very few studies have investigated metrics that can be utilized to determine the testability of a production class. To the best of our knowledge, [7] is the first article that tries to study this matter; it shows that there is a correlation between production code metrics (such as Lines of Code, FANOUT, and Response for Class) and test case metrics (such as Lines of Code for Test Class and Number of Test Cases). Similar ideas are presented by Zhou *et al.* in [8], who demonstrate a connection between testability and structural metrics. However, none of the publications that we have encountered thus far have tried to assess the testability of a production class based on its corresponding tests. We argue that the quantity and quality of the unit tests that cover a particular class are good indicators of how difficult it is to test the respective class. For example, if a production class is addressed by fewer tests compared to other similar classes, then this might suggest that it is more difficult to test.

Two other software quality aspects that are also related to testing are change- and defect-proneness. Change-proneness is a characteristic of software artifacts that represents their susceptibility to modifications; the changes may have various causes, including: 1) new requirements, 2) fixing problems in the code, and 3) ripple effects. There are several publications that address the negative consequences of having high change-proneness. For example, in [9] the authors prove that a lot of maintenance needs to be performed on change-prone classes as a system evolves. Change-proneness has also been associated with technical debt, as demonstrated in [10]. The lack of applying patterns and the presence of anti-patterns have been shown to make a production class change-prone [11].

Another common problem in complex software projects is that they are susceptible to errors [12]. These errors occur because of the high defect-proneness of the system's production classes. There are studies (such as [13], [14], or [15]) that try to assess error-proneness based on software metrics. However, [16] has proven that metrics alone are insufficient to predict defect-prone classes as systems evolve. We argue that the presence of certain design flaws in the production code could make the respective classes more susceptible to faults.

Little research has been done thus far on specific problems in the production code that have a negative effect on the 3 quality aspects discussed above. In terms of testability, [17] presents 4 categories of design flaws that make a system difficult to test. The ones that appear to have the highest impact are those related to class dependencies, namely global state (and singletons) and instantiations that occur in constructors or methods. Design flaws have also been shown to impact change- / defect-proneness. Reference [18] tries to compile a list of flaws that make a class susceptible to change. As mentioned by the authors, this list is by no means complete; further investigation is needed on design flaws that impact change-proneness. Problems in the production code have also been used to predict whether or not a class will change in the future [19]. Therefore, it is even more important to determine other design flaws that make a class more likely to be modified.

Error-proneness is another quality aspect that has received a significant amount of attention in recent years. However, just as for the previous 2 aspects (testability and change-proneness), the impact of design flaws on this quality aspect has not been thoroughly investigated. While there are some studies that look into this (such as [20] or [21]), most of them focus on software metrics rather than on specific problems in the code. In [22] the authors study 5 flaws and establish that there is a correlation (although not very significant) between 4 of them and defect-proneness. Reference [23] presents a literature review of design flaws that may cause software bugs. The 18 studies included in this review only cover around 30 design flaws; this further proves that there are many other flaws that still need to be investigated.

With the exception of [17], none of the other publications consider design flaws related to the usage of the *static* keyword. We named this kind of instances static constructs and will refer to them this way throughout the rest of the thesis. These constructs have already been proven to have a detrimental effect on several other quality aspects, including maintainability [25], understandability [26], and security [27]. For example, [28] presents the most common cases in which the *static* keyword is used in the code and gives a number of reasons why it has a negative effect on maintainability. As mentioned before, Hevery showed that mutable global state and static methods have an impact on class testability. However, the effects of other types of static constructs (e.g., constants or static initialization blocks) have not been studied. We have already proven that 1) mutable global state (static non-final attributes and stateful singletons) negatively affects defect-proneness [29] and that 2) singletons and certain types of static methods make the classes that utilize them more difficult to test [30]. Up until now, we analysed these aspects in isolation; in this thesis we plan to investigate every category of static constructs both in terms of presence / usage and regarding their impact on the 3 quality aspects.

Based on the above, there is a clear need within the scientific community to study the different types of static constructs present in the production code. While some may prove harmless to the software quality aspects that we are addressing, there will surely be categories that have a negative impact on testability or change- / defect-proneness. For example, we expect constants to have little or no influence; on the other hand, stateful singletons should be detrimental to all 3 quality aspects. In order to fully understand how these static constructs are used, we do not think that it is sufficient to examine only the latest version of a system. Valuable insight might be obtained by studying multiple versions throughout a project's history. Therefore, we plan to extract and leverage historical data to further refine this analysis. After we gain a thorough understanding on this matter, we want to study the impact of each type of static construct on the quality aspects mentioned above. By doing this, we will be able to pinpoint the ones that cause problems and provide appropriate recommendations on which static constructs should be avoided.

1.2. Research questions

In this study we try to understand 1) how static constructs are used in complex projects and 2) whether or not they have a negative effect on several software quality aspects. By static constructs we are referring to a broad category of entities that use the *static* keyword. They can be very simple, such as static attributes (non-final and constant) and methods, or more complex (e.g., singletons or utility classes); therefore, an initial categorization is required. Afterwards, we want to study their presence / usage both for the latest version of a system and for multiple versions throughout its lifespan. This is done in order to observe if the usage patterns have changed over the years. As an example, Singleton was initially considered a creational design pattern; however, experience has proven that it is rather an anti-pattern. Thus, we expect such instances to appear less frequently in the final version analysed compared to previous ones. If static construct instances are actually used less, then we need to understand the reasoning behind such a decision. The main cause would be that static constructs are detrimental to software quality aspects. Some of the aspects, such as maintainability or understandability, have already been investigated. We will focus on the ones that have not been addressed thus far, namely: 1) testability, 2) change-proneness, and 3) defect-proneness. For each of these aspects we want to define models that can be used to quantify them. Only after we are able to evaluate a quality aspect for a specific part of the production code can we establish if the parts that contain / utilize static constructs are more problematic than the rest of the code. We plan to analyse the impact of each category of constructs on the 3 quality aspects of interest. This will allow us to specify which types of instances are the most detrimental to a particular aspect. We expect some of the static constructs (e.g., constants) to not have any negative effect on software quality. On the other hand, there might be others (e.g., singletons) which do not impact a quality aspect directly, but rather the production classes that use them are affected. This is also something that we will be considering in our analyses. All of the above have led us to the following research questions:

- RQ1. Are static constructs used in complex software systems?
- RQ2. How have static constructs evolved throughout the lifespan of a project?
- RQ3. Do static constructs have a negative impact on software quality aspects?

With the first question we are trying to establish whether or not static constructs are present in the production code. The following 2 research questions would not make sense if static construct instances do not appear or are barely used. However, we want to distinguish between different types of static constructs. We do not believe that instances of different types are utilized in the same way, therefore we need to categorize them first. The categorization is done based on 1) the size of the construct (e.g., entire classes such as singletons or utility classes vs. a single static method within a class) and 2) whether they represent / access state or not (e.g., static non-final attributes vs. constants). After dividing them into categories, each type will be studied in isolation for the latest version of a system. Some quality aspects might not be directly affected by the presence of static constructs. For example, the singletons themselves are easy to test, but the production classes that utilize them are significantly harder due to the setup required to configure the appropriate singleton state. Thus, the client classes for each instance also need to be considered in the analysis.

The second research question addresses the evolution of static constructs. We want to determine how instances of each type have evolved throughout a project's history. More specifically, we are interested in observing if more instances of static constructs are present / utilized currently compared to the early stages of development. We consider that if static constructs appear less frequently nowadays then this is a clear indication of the fact that they are detrimental to different software quality aspects. Just as for the previous research question, we will also be examining the client classes for each instance. If the number of clients starts to decrease while the system is continuing to grow, then this would further confirm that static constructs are harmful.

Finally, the last research question is directly related to the 3 software quality aspects that we are addressing in this thesis. It can be split into 3 sub-questions; one of them would be: "Do static constructs have a negative impact on class testability?". Therefore, we will investigate the effect of each type of static construct on the 3 quality aspects: 1) testability, 2) change-proneness, and 3) defect-proneness. In order to be able to do this, we must first quantify these aspects for specific parts of the production code. Models and procedures that can aid us in this regard will be proposed. For the classes of interest, the assessments will be performed in relation to other classes which are similar to them (in terms of size and complexity). As an example, we will be capable of establishing if the classes that utilize singletons are more prone to error compared to other classes. By demonstrating that the usage of static constructs is detrimental to one or more of the quality aspects investigated, we will raise awareness regarding the types that are the most problematic.

1.3. Relevance

From a research perspective, it is important to gain insight into the way in which static constructs are used and how they evolved. This allows for a better understanding of software development practices. Additionally, the proposed approach could be employed to study these aspects for other design flaws, such as God Class or Shotgun Surgery. It would be interesting to see if these flaws evolved differently compared to the static construct instances. After obtaining a good understanding on static construct usage, we also want to investigate their effect on 3 software quality aspects.

The aspects considered in this study, testability and change- / defect-proneness, are closely related to the testing process and may affect it. The main goal is to determine which types of static constructs have a large negative impact on the quality aspects studied. By understanding this, we will be able to provide a series of recommendations on which static constructs can continue to be used during development (e.g., constants) and which should be avoided (possibly at all cost). However, the knowledge obtained will not be limited to these aspects. For example, when assessing testability, we want to determine particular smells that exist in the tests covering the classes with static constructs and their clients. By doing this, we can find correlations between certain test smells and the static constructs that cause them to appear. For example, we expect the General Fixture smell to occur more frequently in test classes that cover singleton clients. For change- and defect-proneness we also want to understand the exact modifications that were performed on the classes with / that utilize static constructs. It might be the case that only some specific types of changes occurred and it would be very useful to find out which. The effects of static constructs on other software quality aspects can be studied in a similar way; the appropriate models have to be defined and then the impact of each type of construct can be studied independently.

By investigating all the aspects mentioned above, we will bring a number of contributions:

- A general methodology that can be followed to detect specific design flaws in the production code, study how they evolved, and assess their impact on a series of software quality aspects. This methodology consists of several steps: 1) defining the detection strategies for all the instances of the flaw (possibly categorizing them first); 2) going through the version history of a system to understand their evolution; 3) defining the models for quantifying each quality aspect; 4) comparing the parts of the code in which the design flaw instances are present with other similar classes with regard to each of the analysed aspects. Significant data will be obtained after each step, but only by implementing all of them can we examine the entire process. We will highlight the applicability of the proposed approach with different types of static constructs as the flaws of interest. Three quality aspects are going to be studied: 1) class testability, 2) change-proneness, and 3) defect-proneness. Each category of instances shall be investigated independently in terms of presence, evolution and impact. Afterwards, we will make some general observations regarding static constructs as a whole.
- A model for assessing the testability of production classes based on their corresponding unit tests. The test suite is analysed both from a quantitative and from a qualitative perspective. We rely on coverage data to evaluate quantity, namely line coverage and number of unit tests which address that part of the production code. For quality we detect smells in the

corresponding tests and establish whether or not they are more frequent in the classes that cover static constructs and their clients. The 2 assessments are combined into a score that represents the testability of a certain class. By comparing this score to that of similar classes, we can specify if a class with / that uses static constructs is less / more difficult to test.

- A method to 1) determine the modifications that were performed during a commit and 2) categorize commits as bug-fixes. The first part is needed to establish whether or not the classes with static constructs and their clients 1) were modified more frequently and 2) more changes were performed on them per commit; if this is the case, we can consider them more change-prone compared to other classes. For defect-proneness we investigate the same aspects, but only the commits that were identified as bug-fixes are included in the analysis.
- A tool that can be utilized to detect design flaws, study their evolution and quantify their impact on the 3 quality aspects discussed above (testability, change- and defect-proneness). This tool needs to be as modular as possible; there will be different types of modules for each of the steps from the proposed approach. For example, there are going to be several modules in which we define the detection strategies for every category of static constructs. Another module shall be responsible for retrieving the historical data necessary for studying evolution. Finally, the tool will have a group of modules for assessing each of the software quality aspects. The aforementioned modules can be combined to form the required analysis. The tool should also be highly extendable, new modules with detection strategies or models for other software quality aspects should be added without too much effort.
- An empirical study in which we use this tool for different categories of static constructs. First, we must define the appropriate detection strategy for each type (e.g., stateful / stateless singletons). Then we can analyse their presence / usage both for the latest version of a system and for monthly commits. Finally, we shall use the proposed models / procedures to determine whether or not instances of a certain type have a negative impact on the quality aspects investigated. Through this empirical study we will obtain a good understanding of 1) how static constructs are utilized, 2) the way in which they have evolved, and 3) their effect on the 3 software quality aspects. Some interesting observations are going to be made; they will be discussed in depth in the chapters that follow.

We have set several objectives that must be accomplished in order to provide the contributions presented above. The main objectives of this thesis are:

- O1. Studying the state of the art for the topics of interest: design flaw detection (with an emphasis on static constructs) and evolution, models for quantifying software quality aspects, and design flaws that have an impact on the aspects we are investigating.
- O2. Categorizing the static constructs and defining detection strategies through which instances of each type can be identified. Additionally, analysing the presence and usage of these instances both for the latest version of a project and throughout its entire lifespan.

- O3. Developing procedures through which the quality aspects considered, class testability, change- and defect-proneness, can be evaluated. Also establishing whether or not the static constructs from each category have an effect on them.

1.4. Outline

In this section we explain how the rest of the thesis is structured. The following chapter discusses related work from fellow researchers. It contains 4 sections that cover: 1) different approaches for identifying design flaws; 2) methodologies for analysing the evolution of specific parts of the production code; 3) ways of assessing software testability and change- / defect-proneness; 4) design flaws that have been proven to have a negative impact on these quality aspects. We end this chapter with a section that thoroughly discusses the differences between our work and the other publications with regard to: 1) design flaw detection; 2) studying software evolution (with an emphasis on the design flaws of interest, namely static constructs); 3) quantifying testability and change- / defect-proneness; 4) tools for investigating one or more of the previous aspects.

In Chapter 3 we detail the proposed approach. First, we explain how static constructs were categorized and present the detection strategy for each type. Then we describe the process through which we study the evolution of static constructs. The following sections discuss the model for quantifying class testability and the methods for assessing change- / defect-proneness. We conclude the chapter by providing implementation details for the entire data collection process and presenting the tool that was developed.

Chapter 4 explains how the empirical study was conducted. It starts by discussing the main goal of the study, the formulated hypotheses, and the independent and dependent variables for each hypothesis. Afterwards, we present the criteria based on which we selected the systems included in the study. Finally, we describe in detail each of the 4 analyses that were performed, namely: 1) static constructs presence / usage; 2) evolution of each static construct type; 3) impact on class testability; 4) impact on change- / defect-proneness.

Chapter 5 presents the results that were obtained for each of these analyses. It only includes raw results; their interpretation is provided in the following chapter. In Chapter 6 we revisit each research question and discuss the implications of the results. We also mention a series of threats that might impact the validity of the empirical study and explain how we tried to mitigate them.

The final chapter of the thesis contains conclusions and future work directions. We begin by reiterating the contributions provided through this thesis. Then we summarize what has been done and discuss the main results in connection with the research questions. In the following section we reflect on our work and explain what could have been done better. We end the thesis with 6 future work directions that are being considered at this moment, namely: 1) improving the empirical study; 2) analysing other design flaws; 3) enhancing the testability model; 4) refining the process for identifying bug-fix commits; 5) studying everything at a lower level of granularity; 6) proposing refactoring solutions for static constructs and test smells.

In summary, this chapter discusses:

1. The problem that is being addressed in the thesis; it includes:
    - The importance of testing and the software quality aspects related to this process;
    - The motivation behind choosing static constructs as the design flaws of interest;
    - A short overview of how we plan to tackle the problem.
2. The research questions that were formulated:
    - RQ1. Are static constructs used in complex software systems?
    - RQ2. How have static constructs evolved throughout the lifespan of a project?
    - Do static constructs have a negative impact on software quality aspects?
3. The relevance of our work along with the main contributions:
    - A methodology for studying design flaws, their evolution, and the impact they have on software quality;
    - A model for evaluating class testability based on the quantity and the quality of its corresponding unit tests;
    - A process for determining the fine-grained source code changes performed during a commit and establishing whether or not the respective commit is a bug-fix;
    - A tool that incorporates all of these aspects;
    - An empirical study through which we answer the proposed research questions.
4. How the rest of the thesis is structured:
    - Chapter 2 contains a comprehensive literature review of the articles that address topics which are similar to ours;
    - Chapter 3 describes the approach proposed in order to study the aspects of interest;
    - Chapter 4 presents the empirical study that was conducted;
    - Chapter 5 highlights the obtained results;
    - Chapter 6 provides an interpretation of the results in relation to the research questions along with potential threats to validity that might influence them;
    - Chapter 7 has the conclusions and future work directions.

2. RELATED WORK

As explained in Chapter 1, we need a thorough understanding of what has already been done in terms of design flaw detection and evolution, models for quantifying software quality aspects, and flaws that affect these aspects. Towards this end, we surveyed the literature in order to identify similar work from fellow researchers; the rest of this chapter is structured as follows: i) Section 1 discusses detection strategies and tool support for identifying design flaws; ii) Section 2 explains how the evolution of different parts of the source code has been studied thus far; iii) Section 3 describes procedures that have been utilized to evaluate the software quality aspects of interest; iv) Section 4 covers design flaws the negatively impact the aforementioned quality aspects.

2.1. Design flaw and test smell detection

2.1.1. Detection strategies and tool support

Identifying design flaws is a key activity when trying to achieve the goal specified in the previous chapter. In [31] Brown *et al.* introduce anti-patterns and discuss ways in which they can be detected. The authors define an anti-pattern as "a commonly occurring solution to a problem that generates decidedly negative consequences". They group them in three categories: software development anti-patterns, software architecture anti-patterns and software project management anti-patterns. This study will focus on anti-patterns from the first category, as we will analyze the production code of software systems. For each of the anti-patterns mentioned the authors explain the problem, list the symptoms by which it can be identified and discuss its consequences. This publication can serve as a guideline for understanding a specific anti-pattern and provide a basis for developing the detection techniques necessary for identifying it.

In [32] Marinescu presents a metrics-based approach for detecting design problems and describes concrete techniques that can be used to detect two well-known design flaws, God Class and Data Class. The approach consists of four steps: i) a quantitative analysis of the design-flaw used to define a detection strategy, ii) metrics selection used to express the detection strategy as a combination of metrics, iii) detection of suspects used to obtain a list of code fragments that might be affected by the design flaw, and iv) examination of suspects used to decide whether or not those fragments are actually affected by the flaw. Based on the proposed approach the author defines detection techniques for the two flaws. An industrial case study was conducted in order to prove that the detection techniques can successfully identify the God Class and Data Class design flaws.

The work is continued in [33] which provides a more in depth analysis on design flaw detection based on metrics. The approach is validated through more than ten detection strategies and adequate tool support is provided. The ProDetection toolkit is introduced which supports code inspections based on the detection strategies defined. The process consists of three steps: i) creating a meta-model of the software system, ii) running the detection strategies in order to obtain a list of suspect design entities, and iii) manually verifying the results. The usefulness of the toolkit is demonstrated through an industrial case-study which also highlights the accuracy of the detection strategies that were defined.

Reference [34] also introduces a tool (HIST) that can identify five design defects and provides an analysis on when they appear throughout the lifespan of a system. Besides structural information, this tool also leverages co-changes extracted from versioning systems to detect the following flaws: Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob and Feature Envy. For each of the smells a historical detector is defined using a combination of association rule discovery and by analyzing the set of classes/methods that are co-changed. Two empirical studies were conducted in order to evaluate HIST. The first assesses its recall (between 58% and 100%) and precision (between 72% and 86%) on twenty open-source Java projects. The second study involved twelve developers of four open-source systems who concluded that more than 75% of the problems identified by HIST are actual design flaws.

In [35] Kessentini *et al.* go deeper than simply detecting design flaws, they also enable the refactoring of the analyzed code in order to remove them. The proposed approach utilizes Genetic Programming to automatically generate rules for detecting design flaws. Afterwards, a Genetic Algorithm is used to generate refactoring solutions that can be applied to remove the flaws that were identified. The approach is validated using six open-source software systems. The results show that more than 76% of the design flaws were successfully detected and the correction solutions proposed were able to remove 74% of them.

Besides detection strategies [36] also proposes a rigorous process (based on precision and recall) that can be used to validate the strategies. The main contributions of the article are: i) a method that can be utilized to define the steps required to detect design flaws (DECOR), ii) a detection technique that instantiates this method (DETEX), and iii) an empirical validation of the detection technique. DETEX can be used to identify four design smells (Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife) and their underlying code smells. It was validated on eleven open source systems and showed precision of over 60% and 100% recall. The complexity of the obtained detection algorithms and the computation times required are also discussed, both of which look reasonable.

In [37] Wegrzynowicz and Krzysztof present an approach for building test suites for detectors of design patterns. The usefulness of the approach is proven by creating a test suite containing a set of implementation variants for the singleton pattern. Afterwards, three pattern detectors were evaluated (in terms of accuracy) using this test suite and it was shown that each of them had their limitations in detecting all the variants correctly. A similar approach could be used to validate design flaw detectors.

2.1.2. Test smells

The presence of design flaws in the production code might indicate that there are also problems in the test code. These problems are generally referred to as test smells, they represent deviations from the guidelines that were proposed to aid developers in creating good test suites. Reference [38] is one of the first papers to introduce test smells and proposes solutions for detecting two of them, General Fixture and Eager Test. The authors present the characteristics of a good test based on the core principles of unit testing. Then they discuss the structural deficiencies that cause test smells to appear (in terms of test concepts and their characteristics). Based on this a set of metrics is defined that can be used to identify the two test smells mentioned above. An initial validation of the detection strategies is done using manual inspection and code review. In [39] Bavote *et al.* present two empirical studies on test smells. The first analyzes the distribution of smells within the test base. The study includes eighteen systems (two industrial and sixteen open source) and shows that test smells are widely spread throughout these systems (in 82% of the test classes). The second study investigates their impact on program comprehension during maintenance and testing and proves that there is indeed a negative impact.

The authors went more in depth in [40] which proves that test smells are harmful and occur frequently in software systems. Similar to the previous paper, two empirical studies were conducted. The first proves that code smells are present in both open source and industrial systems with 86% of unit tests having at least one test smell present. The second study shows that test smells have a negative impact on maintenance and program comprehension. Its main finding was that comprehension is 30% higher if test smells are not present.

In [41] Jianping *et al.* present an empirical study on the relationship between test smells and production class features. The study investigates whether the complexity properties of a class can be used to predict test smells in its corresponding unit tests. It was conducted using five open source systems and found that Cyclomatic Complexity (CC) and Weighted Methods per Class (WMC) are indicators of the presence of test smells, predominantly Eager Test and Duplicated Code. Other production class features were also considered; the Lack of Cohesion of Methods (LCOM) also correlated with the two test smells while the Depth of the Inheritance Tree (DIT) did not.

Reference [42] is a PhD thesis that addresses software testability and the quality of testing in object-oriented software systems. It contains several important findings including the fact that there is a correlation between certain code smells and test smells. The study also shows that unit tests are not distributed in line with the system's dynamic coupling. Many of the tightly coupled classes do not have associated unit tests, while the loosely coupled ones have at least one direct unit test. Furthermore, the results highlighted that there is a correlation between class testability and dynamic complexity. A larger number of unit tests is required to address classes that are executed more frequently within the code.

2.2. Software evolution

Historical data have already been used to improve design flaw detection [43]. We also utilized this kind of information to study the co-evolution between production and test code [44]. We did this in order to obtain a better understanding of the way in which tests evolve as a result of changes in the production classes. An association rule mining technique was used to uncover 6 fine-grained co-evolution patterns. We also established that the testing effort that is put into a project does have an impact on the observed patterns.

The lifespan of code smells is studied in [45]; the authors investigate the behavior of the developers with regard to the removal of code smells and anti-patterns. Their results indicate that software engineers are aware of the presence of these flaws, but are not too concerned about their impact on quality. This observation is further supported by [46], which found that code smell removal with refactoring tools is often avoided during maintenance.

One of the first attempts to study the evolution of a particular design flaw (God Class) is presented in [47]. The histories of 2 open-source projects are analyzed in order to establish how God Classes appeared, their prevalence, and whether or not they are still part of the system or were removed during its evolution. The authors also manage to distinguish between the God Classes that were created by accident and those that are so by design.

The evolution of design smells along with their impact on the change behavior of software systems is evaluated in [48]. The proposed analysis identifies "good" and "bad" phases in the evolution of a system, which correspond to decreases / increases in the number of components that contain smells. It also proves that the respective classes have a higher change frequency. However, only 2 flaws, God Class and Shotgun Surgery, are considered and the study includes just 2 open-source systems. Five other smells, Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, and Spaghetti Code, are studied in a similar manner in [49]. The paper is aimed at understanding when and why the code starts to smell bad, but (just like the previous study) only addresses a limited number of smells from a small number of systems.

We will strive to obtain an analysis as thorough as the one presented in [50], but with a lot more design flaws and systems. Tools that could assist us in this regard have started to emerge. As mentioned before, [34] introduces HIST which can identify 5 design defects and provides an analysis on when they appear during the lifespan of a system. Another example would be Coming [51], a tool that can be utilized to mine instances of change patterns.

2.3. Evaluating software quality aspects

2.3.1. Assessing software testability

Considering that our research addresses improving the testability of object-oriented systems, there is a need to develop a model by which this aspect can be quantified. In [52] Mouchawrab *et al.* describe one of the first attempts to measure the testability of software systems. The proposed framework does not start from the source code but from the UML diagrams that model the system. The authors also introduce a set of attributes that can have an impact on testability; they group them in three categories: i) Object Constraint Language Expression Complexity, ii) Use Case Model and System Interface Complexity, and iii) Interactions Between Inherited and Overridden Features. Measurement procedures are provided for each of the attributes. The authors also discuss ways in which these measurements can be interpreted in order to assess the testability of the system.

Reference [53] is the first publication to investigate code metrics that can be utilized to quantify testability. Five Java systems (one open source and four industrial) were analyzed using the GQM/MEDEA framework. It was found that there is a correlation between production code metrics (such as Lines of Code, FANOUT and Response for Class) and test case metrics (such as Lines of Code for Test Class and Number of Test Cases).

Similar ideas are presented by Zhou *et al.* in [54], who show that there is indeed a correlation between testability and software structure metrics. The main findings of this empirical study are that: i) metrics related to size, complexity and coupling have a higher impact on the testability of a software system compared to other production code metrics (related to cohesion or inheritance), but ii) metrics alone cannot predict the effort required for unit testing a class. The study also demonstrated that it is better to apply partial least square regression (PLSR) than multiple linear regression (MLR) when trying to correlate metrics with unit testability.

Reference [55] contains a survey on models that can be used to determine the testability of an object-oriented software system. For each of the analyzed models the authors discuss the method used to assess testability along with the achievements of the model and its main issues. They conclude that "there is no single superior model", thus a testability model should be chosen based on the particularities of the analysis you are trying to conduct. All of the surveyed models try to assess testability during the design and analysis phase, they do not address already implemented code.

2.3.2. Assessing change- / defect-proneness

Just as for testability, the first studies that address change-proneness try to evaluate this software quality aspect based on the design of a system. For example, [56] proposes a method for calculating the behavioral dependency measure (BDM), a metric that can be utilized to predict change-proneness. The work is continued in [57], where the authors present a case study in which they evaluate the usefulness of this metric using a multi-version open source project called JFlex.

There are several articles that investigate the capability of object-oriented metrics to predict change-proneness. However, most of them only take into account a small amount of OO metrics or study a limited number of systems. In [58] the authors analyze 102 Java projects and assess the effect of 62 metrics. Their results show that size metrics have the highest ability of predicting whether or not a class is susceptible to modifications. For the coupling and cohesion metrics the capacity is lower, while inheritance metrics cannot be used to distinguish between classes that are change-prone and those which are not.

Other studied use machine learning techniques to predict change-prone classes. The effectiveness of such approaches is evaluated in [59] and compared to that of statistical techniques. The article proves that both types of methods can be used to evaluate this software quality aspect. It also highlights a series of OO metrics that are more suitable in this regard.

Reference [60] investigates the relationship between design patterns / meta-pattern roles and change-proneness. The authors also study the effect of the size of the classes on this software quality aspect. Their results show that the latter has a much larger impact on change-proneness compared to the patters and meta-pattern roles.

Object-oriented metrics have also been used to assess software defect-proneness. For example, the effects of size on this software quality aspect have been investigated in [61]. Cox proportional hazards modeling with recurrent events is utilized for the assessment; however, only 1 project (Mozilla) is used in the evaluation.

A considerably more complex model for evaluating defect-proneness is proposed in [62]. It includes 3 types of bad design indicators, including several code smells, high method dependency, and large file size. The study shows that each of these types has a negative effect on defect-proneness and that these effects are relatively independent of one another. This is an important observation considering that we plan to investigate the impact of a specific category of design flaws (static constructs) on this software quality aspect.

There are few publications that try to leverage historical data for defect prediction. As an example, in [63] Moser *et al.* categorize Java classes as defective or defect-free based on two sets of metrics, product related and process related; they prove that the metrics from the second category are more efficient for predicting errors. This observation is further supported by [64]; it shows that code metrics do not change that much from one version to another, thus leading to the stagnation of the defect-proneness prediction models.

2.4. Design flaws that affect software quality

2.4.1. Testability

Design flaws have an impact on various aspects of software systems including understandability [65] or maintainability [33]. Our study complements existing ones and focuses on their impact on testability and change- / defect-proneness. In [66] Hevery presents several design flaws that make software systems difficult to test. They are grouped into four categories: Constructor does Real Work, Digging into Collaborators, Brittle Global State and Singletons, and Class Does Too Much. For each of these categories the author discusses the reasons why they have a negative impact on testability, a series of warning signs and ways in which they can be fixed. Additionally, concrete examples are provided that allow for a better understanding of the underlying problems along with possible refactoring solutions.

Reference [67] describes a tool that was developed based on the concepts introduced in [66]. The tool can be used to analyze software systems and generate a testability report. This report contains information regarding the design flaws that affect a system's classes along with scores that quantify each flaw's impact on the testability of a certain class. Besides the testability evaluation the tool also provides concrete refactoring solutions that can improve the overall testability of a system.

In [68] Sabane *et al.* investigate the effects of anti-patterns on the cost of class unit testing and propose a number of refactorings that can reduce this cost. The indicator of testing cost considered was the number of test cases that satisfy the minimal data member usage matrix (MdMUM) criterion. A study was conducted using four open source systems which showed that classes that contain design flaws require a higher number of unit tests compared to other classes that are not affected by flaws. It also highlighted that the testing cost can be significantly reduced by refactoring the classes to remove the design flaws. An additional finding was that certain flaws (such as Blob, Anti-Singleton or Complex Classes) have a higher impact on this cost compared to others (such as Method Chain or Lazy Classes).

Reference [69] introduces the concept of testability anti-patterns and discusses two configurations of an object-oriented design that have a negative impact on its testability. These anti-patterns appear when "potentially concurrent client/supplier relationships between the same classes along different paths exist in a system”. The paper also discusses testability issues that might arise when applying certain design patterns. Based on this the testability grid was created which can serve as a guideline on the risk of applying a certain pattern. The authors also defined a series of testability constraints that can help minimize this risk.

Design flaws are not the only factors that have an impact on the testability of software systems. In [70] Tahir *et al.* present an empirical study on the degree of association between runtime properties and class-level testability of object-oriented systems. Similar to our work testability is evaluated at the unit test level; two measurements are used to characterize it, size (test lines of code) and intended design (number of test cases). The results prove that there is a correlation between Dynamic Coupling and Key Classes and the testability of a class. Some of the Dynamic Coupling metrics utilized (such as Export Coupling) had a stronger correlation with the two testability measurements than others (such as Import Coupling).

Other properties of software systems can make them difficult to test. Reference [71] introduces the concept of test-critical dependencies and proposes an approach that can be used to identify them. They are “dependencies within a system that are critical to test complexity” therefore should have an impact on testability. The main findings were that a small number of dependencies have a high impact on the testability of a system and that conventional coupling metrics cannot be used to identify them.

2.4.2. Change- / defect-proneness

The impact of several design flaws on software defects is discussed in [72]. The flaws considered are: Brain Methods, Feature Envy, Intensive Coupling, Dispersed Coupling, and Shotgun Surgery. The results showed that although the flaws do correlate with software defects, it was impossible to determine which ones are the most harmful. They also proved that an increase in the number of design flaws makes a system more susceptible to errors.

Reference [73] assesses technical debt based on the flaws present in a particular version of a system. A framework is proposed and its effectiveness is proven by analysing the evolution of technical debt symptoms and uncovering past refactoring actions. This study shows that these refactoring actions are not always organized and coherent, not even when experienced developers are involved.

A study on the effects of anti-patterns on change- and fault-proneness is presented in [74]. However, there are several key differences between this research and what we are going to do: 1) from the 13 anti-patterns investigated only one is overlapping with our work, namely stateful Singleton; 2) only major releases are considered for the studied systems, while we will adopt a more fine-grained approach (analyse all the commits); 3) the methods for quantifying change- / defect-proneness are different in terms of versioning system, metrics, and categorization (e.g., bug-fix commits).

Similar observations can be made with regard to [75]. While this study is more elaborate than the previous one, it does not consider the category of flaws that we will focus on (static constructs) or the ones we are planning to address in the future (e.g., object instantiations in constructors / methods or Law of Demeter violations). Furthermore, we want to investigate a large variety of quality aspects, not just the ones related to maintainability.

There are also publications which suggest that design flaws have a very limited effect on certain software quality attributes or no impact at all. One such example would be [76] in which the authors conclude that the effect of smells on the overall maintainability of a system is relatively minor. Reference [77] also establishes that the impact of the 12 design smells that were investigated on maintenance effort is small. Similarly, [78] proves that even though some smells do have an effect on fault-proneness this effect is quite limited. All of these aspects are worth investigating for other design flaws that have not been considered until now (such as testability).

2.4.3. Class dependencies

By studying the design flaws that make object-oriented systems difficult to test or more change- / error-prone, it was found that those related to class dependencies have one of the biggest negative impacts [79]. This includes global state (and singletons) and instantiations that occur in constructors or methods, so we have decided to investigate these flaws in greater depth.

In [80] Hevery explains why static methods have a negative impact on the testability of software systems. He states that the main problem with this type of methods is the fact that they represent procedural code which is difficult to test. Unit testing assumes that a part of the application can be instantiated in isolation. During instantiation a series of dependencies are put together using mock objects in order to replace the real dependencies, thereby enabling that part of the code to be tested. This is impossible for procedural programming because there are no objects involved, the methods and the data are separate from one another.

Reference [81] presents the most common cases in which the static keyword is used in the code and gives a number of reasons why it has a negative impact on several aspects of object-oriented systems including maintainability and testability. The cases mentioned are: i) worker methods (used for different kinds of processing tasks), ii) factory methods (used to return preconfigured instances of a class), iii) singleton methods (used to limit the number of instantiations to a single global instance), and iv) global variables (used to store various configurations). The reasons why this static code causes problems are violations of the main principles of object-oriented programming (such as encapsulation), encouraging tight coupling between the system's classes and hindering unit testing.

Reference [82] introduces the concept of "Class-Oriented Programming" and explains the main issues with this paradigm. It refers to classes that have only static attributes and methods and are never instantiated. The article discusses problems caused by such classes and concludes that without objects and their interaction it is impossible to build complex software systems. Similar ideas are presented in [83], an article that mentions the only two situations in which using the static keyword does not cause problems, global constants and constructor-like static functions (used to replace overloaded constructors which might become ambiguous). In all the other cases having static members might cause problems because it is unclear in which class they should actually be placed. In general, static methods tend not to use the attributes of the classes in which they reside, thereby leading to violations of the Single Responsibility Principle (SRP). Other reasons mentioned for why static methods should not be used are the fact that they cannot be called polymorphically, they increase the complexity of a software systems and they are difficult to test (especially when new instances of other classes are created within them).

In [84] Feathers proposes a rule that should be followed to make the code easier to test: "Never hide a Test Unfriendly Feature within a Test Unfriendly Construct". The Test Unfriendly Constructs (TUCs) mentioned include static methods, static initialization expressions and static initialization blocks while the Test Unfriendly Features (TUFs) are lengthy computations, accesses to side effecting APIs and database/file system/network accesses. The author also advocates using Test-Driven Development as a means to ensure testability.

The authors of [85] discuss both the benefits and the drawbacks of the Java instantiation mechanism. The benefits mentioned are a clear and comprehensible syntax, the ability to chain constructors in class hierarchies and the fact that correct initialization can be enforced on the class's clients. However, there are also two drawbacks, namely that it is not polymorphic and that it allocates memory from the heap. These do not occur in all of the cases, memory is not allocated when the constructor is called using super and calling constructors through reflection is polymorphic. In the context of testability the second drawback causes major difficulties. Using new to instantiate a specific class in a constructor or method creates a dependency to the concrete type of the class that is being instantiated (because new is non-polymorphic). However, this can be solved through Dependency Injection. The usefulness of DI is also discussed in [86], where Hevery provides a concrete example that highlights why dependency injection is better that object instantiation.

Reference [87] contains two chapters in which Feathers discusses how to deal with global mutable state and object creation in constructors when trying to refactor legacy code. In terms of instantiations in constructors the author specifies adding an additional parameter (instead of the instantiation) as the recommended solution. Overloaded constructors can be used so that not all of the clients are forced to pass the additional argument. Feathers also proposes several solutions that can be applied when testing global state. For singletons a static setter could be added to the class and the constructor can be changed to protected. Afterwards the singleton can be subclassed, an object of the subclass created and passed to the setter. For static attributes making them non-static and passing them as parameters is considered a better practice than accessing them as global variables. All the refactorings mentioned above make the legacy code easier to test thereby providing a solid basis when trying to restructure it.

Feather's book also introduces the concept of seam. The author defines it as "a place where you can alter behavior in your program without editing in that place". The usefulness of seams is highlighted when trying to break dependencies in order to test legacy code. The types of seams differ from one programming language to another; Feathers provides examples of processing seams, link seams and object seams. Considering that in our investigations we will be utilizing systems developed in an object-oriented programming language (Java) only the last two categories of seams are of interest. Java does not have a build stage before compilation therefore processing seams cannot be leveraged. Each seam has an enabling point which is "a place where you can make the decision to use one behavior or another". For link seams the author provides an example where classpath is used as the enabling point that switches between different implementations of a class that is included in the class under test. Calling a method on an object that is received as a parameter by the method in which the call is made was the basic example given for object seams. In this case the enabling point is the parameter as its type can be controlled through the argument that is given to the method during unit testing. The author specifies that object seams are the best choice when trying to get portions of code under test in object-oriented languages. The other types of seams are not as explicit as object seams and can make the tests that depend on them more difficult to maintain.

In [88] Gil *et al.* introduce a catalog of micro-patterns that can be identified in the source code of Java systems. The authors argue that more than 75% of a system's code is written based on these micro-patterns. The patterns are divided into eight main categories, three of which address static members (attributes or methods): Degenerate Behavior, Degenerate State and Controlled Creation. In terms of actual micro-patterns the ones of interest are: Stateless (a class with only static final fields), Common State (a class with only static fields), Restricted Creation (a class with no public constructors and at least one static fields of the same type as the class) and Cobol Like (a class with a single static method and with no instance members). Finally, the authors hint at nano-patterns which are patterns of methods and state that a combination of micro- and nano-patterns can be used to decompose an entire system.

Reference [89] illustrates how Java developers implement and test singletons. Several aspects were taken into account including multithreading, classloaders and serialization. However, the problems that singletons introduce in the context of testability persist. Singletons represent a global and static way of obtaining an instance of a class which makes mocking impossible. Similar ideas are expressed in [90] which discusses the main issues with singletons and possible ways in which they can be removed. The author states that singletons are dangerous because they make debugging and unit testing difficult. The main problems mentioned are that they create dependencies which are hidden within the code (cannot be detected by examining the interfaces of classes or methods) and other classes are tightly coupled with the singleton instance (thus polymorphism cannot be used). These problems could be alleviated using dependency injection, possibly through DI frameworks like Spring or Guice. The main takeaway from the article is that object creation should be separated from business logic and singletons are preventing this.

In summary, this chapter discusses:

1. Different types of approaches that have been proposed to identify design flaws.
2. Methodologies for studying the evolution of certain parts of the source code.
3. Models for quantifying the software quality aspects of interest, namely class testability, change- and defect-proneness.
4. Design flaws that were shown to have a negative effect on these quality aspects.

3. APPROACH

The chapter discusses the approach that was adopted in order to study 1) how static constructs are used (both in the latest version of a system and for previous iterations) and 2) their effects on several software quality aspects. First, we explain how we categorize the static constructs and propose detection strategies for each type. Next, we describe the process through which we study the evolution of the different types of static constructs and the production classes that utilize them. The methods used for 1) assessing the testability of a particular class and 2) identifying change- / error-prone classes are thoroughly discussed in the following 2 sections. We end this chapter by providing concrete implementation details for the entire data collection process; for each decision that needed to be taken we try to explain the reasoning behind it.

3.1. Categorizing and detecting static constructs

As discussed in Chapter 1, we do not believe that all static constructs are detrimental to the software quality aspects investigated. For example, constants should not have a negative effect on the testability of 1) the production classes in which they are declared or of 2) the classes that utilize them (if any). Because they are final, only 1 unit test is needed to determine if the value stored in them is correct. On the other hand, we do think that other static constructs (such as stateful singletons) have a high impact on this quality aspect. While they themselves might not be that difficult to test, the production classes that utilize them may be tested less because of the setup required to configure the appropriate singleton state.

In order to establish which types of static constructs influence the quality aspects studied we must first categorize them. Considering the varied granularities of the constructs in which the static keyword is used (entire classes for singletons vs. small parts of a class for constants or static methods), we decided to perform a multilevel categorization. At class level we distinguish between 3 types of static constructs: 1) singletons (both stateful and stateless), 2) utility classes, and 3) classes that contain only smaller instances. The detection strategies for the first 2 types are as follows:

- for singletons 3 conditions have to be met for the general form:

  1. there are no public constructors within the class;

  2. the class has a private static attribute (the ”singleton instance”) and a public static accessor method that performs lazy instantiation on this attribute and returns it;

  3. the aforementioned method is the only way in which the respective attribute can be accessed.

- for utility classes there are also 3 conditions:
  1. there are no constructors within the class;
  2. the class only has static final attributes (constants);
  3. all the public methods from the respective class are static.

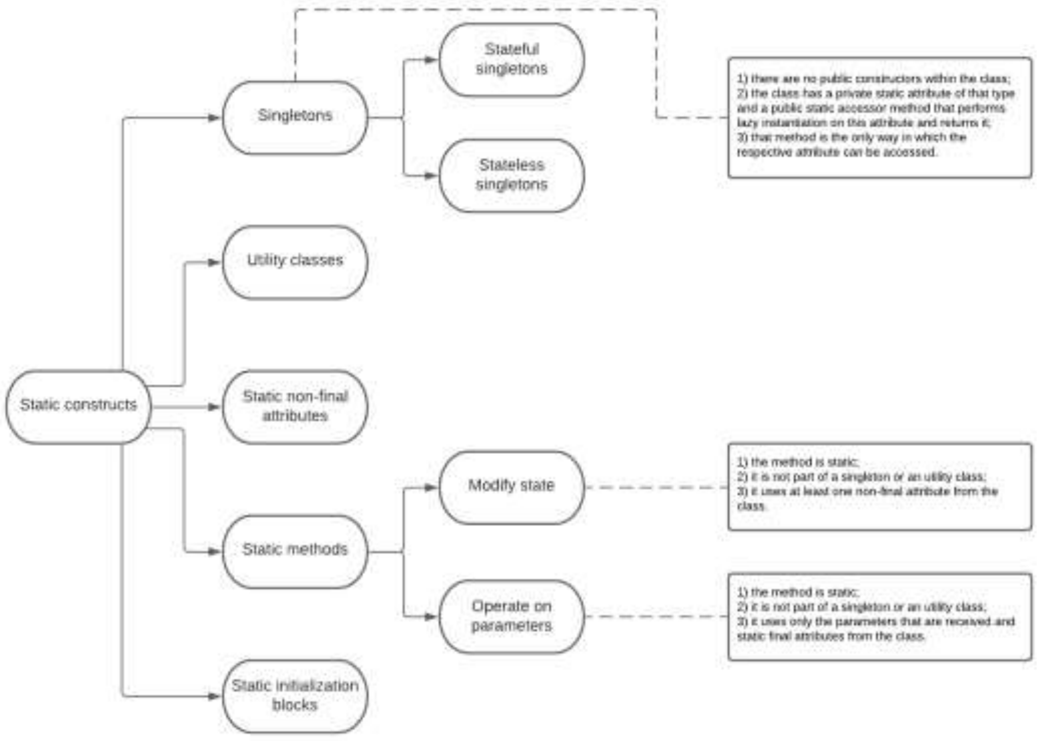
Static constructs
Singletons
Stateful singletons
Stateless singletons
1) there are no public constructors within the class;
2) the class has a private static attribute of that type and a public static accessor method that performs lazy instantiation on this attribute and returns it;
3) that method is the only way in which the respective attribute can be accessed.
Utility classes
Static non-final attributes
Static methods
Modify state
1) the method is static;
2) it is not part of a singleton or an utility class;
3) it uses at least one non-final attribute from the class.
Operate on parameters
1) the method is static;
2) it is not part of a singleton or an utility class;
3) it uses only the parameters that are received and static final attributes from the class.
Static initialization blocks

Figure 3.1. Overview of static construct categories

For singletons the strategy was further refined so that it can detect several variations of the pattern, namely the ones discussed in [91]. Therefore, besides the general, Lazy Instantiation variant, we are also able to identify 7 other forms: Eager Instantiation, Replaceable Instance, Subclassed Singleton, Delegated Construction, Different Placeholder, Different Access Point, and Limiton. By doing this we expect to increase the number of detectable instances, thereby improving the quality of the analysis. An additional condition is required to distinguish between stateful and stateless singletons. For the stateless ones we need to check that the respective class has only constants as attributes.

The classes that contain static constructs but are not singletons or utility classes are categorised based on the types of the instances present. Those that have static methods are divided into 2 categories: a) the static methods utilize / modify the class's attributes and b) the static methods only operate on the received parameters. For these 2 types of instances the detection strategies are:

1. the method is static;
2. it is not part of a singleton or a utility class;

3a. it uses at least 1 non-final attribute from the class in which it is located;

3b. it uses only the parameters that are received and static final attributes from the class.

Finally, there are 3 more types of static constructs included in the categorization. The first 2 are related to attributes, namely static non-final ones and constants. They are easily detectible by going through all the attributes of a class and determining those that are declared static; in order to be categorized as constants they also need to be final. The last type of constructs are static initialization blocks, chunks of code enclosed in braces that are preceded by the static keyword. They are identified by determining the static instances from a class that are neither attributes nor methods. An overview of all the categories of static constructs is depicted in Figure 3.1.

3.2. Studying evolution

We rely on Git to obtain the data necessary for studying how static constructs and their clients have evolved throughout the lifespan of a system. GitHub was chosen because it provides access to numerous repositories for a wide variety of software projects. For each of the analysed systems sampling is performed on their commits with a frequency of 1 commit per month. We consider this time frame appropriate because although it is possible that static constructs were added and subsequently deleted in a single month, we do not think that such rapid changes are meaningful for our analysis. Afterwards, we compute the differences between each commit and the one that was selected for the previous month. We do this for every category of static constructs; these differences include: the total number of instances per category, the number of client classes for each instance, and the average number of clients for the entire project. Additional data related to each static construct and all of its clients from the respective commit are also recorded along with other useful information (e.g., a class being marked as Deprecated). These data are then used in our analysis on the evolution of different categories of static constructs.

3.3. Quantifying class testability

Unlike previous studies that address this software quality aspect, we evaluate the testability of a production class based on the quantity and the quality of its corresponding unit tests. We rely on code coverage data to determine quantity, while for quality we check for certain smells that might appear in the test classes. Coverage information was considered because the lack of code coverage for a specific class in comparison to other similar classes would indicate that the respective class is more difficult to test. We look at 2 aspects when evaluating quantity: 1) the line coverage obtained for a production class, and 2) the percentage of methods from the class that are addressed by unit tests. Although the first aspect would already be a good indicator of how thoroughly a class is tested, we also investigate the second aspect in order to avoid situations in which a limited number of large production methods are adequately covered by tests while all the remaining ones (of a smaller size) are completely disregarded. The coverage data are collected using JaCoCo [92] as it can be utilized on both Maven and Gradle projects and provides a detailed report which also includes some class / method complexity measurements.

Test smells are problems in the unit tests that might negatively affect the quality of test suites, thus also having an impact on the production code that is being addressed. Reference [93] defines them as "deviations from the guidelines that were proposed to aid developers in creating good test suites". The presence of certain smells in the tests that cover classes which have / use static constructs can indicate that they are indeed harmful to testability. Let us consider singletons as an example. The Eager Test smell may appear because several production methods that utilize a singleton are called in the same unit test in order to avoid recreating the specific state needed for the singleton in multiple tests. In the same vein, the General Fixture test smell might be present in the setup method of a test class due to the fact that the state of a singleton is configured in the respective method even though it is only required in some of the unit tests. Both of the above are examples in which using singletons in the production code determines bad practices in the test classes.

In terms of test smell detection, they are identified through *tsDetect* [94]. The tool takes as input a CSV file containing all the test classes of a system along with the production classes they are covering. As output it generates a CSV file that indicates which of the 19 smells are present in each test class. An overview of the test smells that are detected is provided in Table 3.1. Even though some of the smells are quite general (e.g., Empty Test), most of them represent real problems in the test code that may be correlated with a class's lack of testability. Simply determining whether or not a specific smell is present in a test class is insufficient for a thorough analysis on unit test quality. Therefore, the tool was extended so that it can identify which (and how many) smells are present in a particular test.

When assessing the testability of a production class we do not compare it to all the other classes in the system, but rather with similar classes in terms of size and complexity. It would not make sense to compare the testability of a large, complex class (which by its nature is difficult to test) with that of a small, trivial one. To compute similarity, we rely on *Patrools* [95] to extract size metrics (such as lines of code or number of methods) while for complexity we also integrate the scores provided by *JaCoCo*. Now that we have determined groups of production classes which can be considered similar, we need a suitable method for comparing the classes that are part of a group. It would be difficult to reason in terms of individual values (e.g., line coverage or total number of smells present in the corresponding test classes); thus, an aggregated metric is much more appropriate.

To aid us in this endeavour we introduced the testability score. This complex metric combines both the quantitative and the qualitative aspects of the corresponding test code and represents the difficulty of testing a certain class. If a production class has a higher testability score than another, then the latter is harder to test. In order to compute this score, the aspects of interest are assessed independently. As discussed before, for test code quantity we consider 1) line coverage and 2) percentage of production methods addressed by unit tests, while for quality we look at 1) percentage of unit tests in which smells are present and 2) number of different types of smells that appear in a test class. For each of these 4 aspects, a score between 0 and 5 is assigned based on a set of threshold values; the thresholds for each aspect are summarized in Figure 3.2. For example, the corresponding score for line coverage is: 0 if less than 10% of the code is addressed by unit tests; 1 for coverage between 10% and 25%; 2 for coverage between 25% and 50%; 3 for coverage between 50% and 75%; 4 for coverage between 75% and 90%; 5 if more than 90% of the code is covered by tests.

Table 3.1. Test smells identified by *tsDetect*

Test smell	**Acronym**	**Description**
Assertion Roulette	AR	test method has multiple non-documented assertions
Conditional Test Logic	CTL	test method has one or more control statements
Constructor Initialization	CI	test class has a constructor declaration
Default Test	DT	test class has default behaviour (auto-generated by various development environments)
Duplicate Assert	DA	test method has more than one assertion with the same parameters
Eager Test	EaT	test method has multiple calls to more than one production method
Empty Test	EmT	test method does not have a single executable statement
Exception Handling	EH	test method has at least one throw statement or catch clause
General Fixture	GF	not all the attributes instantiated in the setup method of a test class are utilized in every unit test
Ignored Test	IT	test method or the entire test class has an @Ignore annotation
Lazy Test	LT	multiple unit tests from a test class call the same production method
Magic Number Test	MNT	test method has one or more assertions with a numeric literal as an argument
Mystery Guest	MG	test method has object instances of file or database classes
Redundant Print	RP	test method calls one or more write methods from the System class
Redundant Assertion	RA	test method has an assertion in which the expected and actual parameters are the same
Resource Optimism	RO	test method makes an optimistic assumption that an external resource (e.g., a file) is available
Sensitive Equality	SE	test method calls the toString() method in one or more of its assertions
Sleepy Test	ST	test method calls the Thread.sleep() method
Unknown Test	UT	test method does not have a single assertion or @Test(expected) annotation parameter

The scores for the 2 aspects from each category (quantity / quality) are aggregated through a mean value; thus, we compute 2 new scores, one for quantity and another for quality. These 2 values are aggregated once again using the same procedure, thereby obtaining the final score for testability. As an example, we have a production class for which we calculated the following metrics: 1) 57.5% line coverage, 2) 46.5% of its methods are addressed by unit tests, 3) 23% of the tests have at least 1 smell, 4) 4 different types of test smells were encountered. The corresponding individual scores are 3, 2, 4, and 2, respectively. Therefore, the quantitative score for this class is 2.5 while the qualitative one is 3. As a result, the testability score for the class is 2.75. If a similar class has an overall score of 4, it means that the class is easier to test than the one which was analysed. Both the individual scores and the aggregated ones can provide insight into how difficult it is to test a particular class. By determining in which of the investigated aspects the production classes suffer more, we could suggest certain improvements to the testing process.

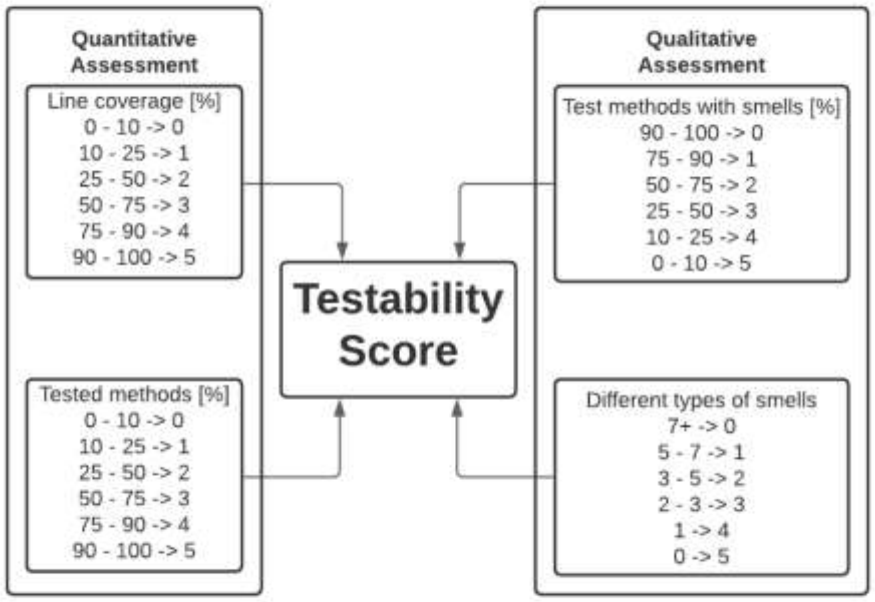

Figure 3.2. Thresholds for the quantitative and qualitative aspects

3.4. Identifying change- / error-prone classes

In order to detect the classes that are more susceptible to changes / errors we have to rely again on a system's version history. The key difference when evaluating these 2 quality aspects is that for error-proneness we only consider the commits that correspond to bug-fixes. Therefore, determining whether or not errors were resolved in a particular commit is the first step in the entire process. Besides the information extracted from the commit message, we also need access to the Jira instance for the respective project to retrieve a list of issue keys corresponding to bugs. The following steps can be followed to establish if a commit is a bug-fix:

1) we check whether or not the commit message contains a Jira issue key;

2a) if it does, we test the key against the list that was computed earlier;

3a) if the list contains the key, we mark the commit as a bug-fix;

3b) otherwise the commit is disregarded as it is related to other development tasks (e.g., adding a new feature);

2b) if the commit message does not include an issue key, we check for specific keywords (such as bug, error, or fix) within the message;

4a) if at least 1 keyword is present along with a production class / method name, the commit is considered a bug-fix;

4b) otherwise it is ignored in the following analysis;

As shown above, both the commit message and the information extracted from the associated Jira issue tracker are used to categorize commits as bug-fixes. First, the commit message is parsed in order to determine if it contains 1 or more

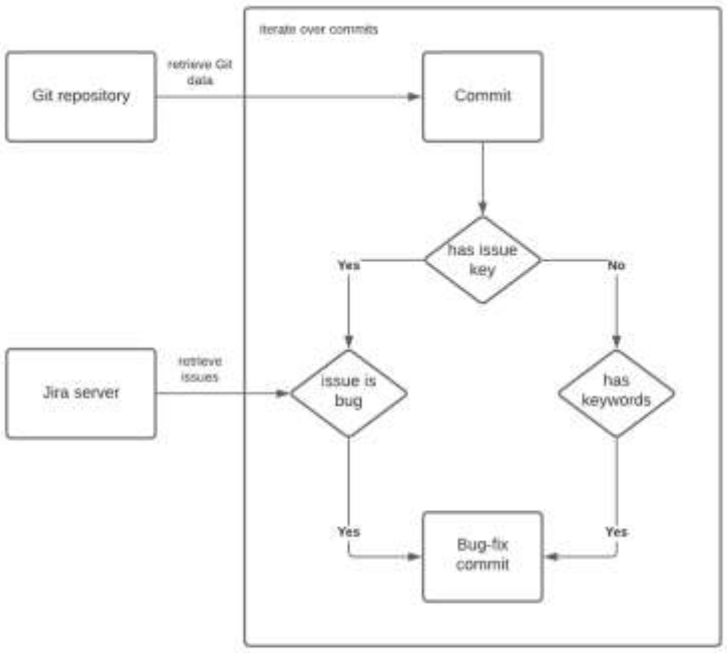

Figure 3.3. Categorizing commits as bug-fixes

issue keys. If this is the case, we check if the respective key corresponds to a bug based on the list of bugs that was computed earlier. The commits with such keys are considered bug-fixes; the others are disregarded because the issue keys correspond to other development tasks, including but not limited to: improvements, new features, auxiliary tasks, and testing. If there is no Jira issue key in the commit message, we check for variations of particular keywords such as bug, error, or fix; in case we find such a keyword, we also look for class / method names. If a class name is encounter, we consider that the respective production class was fixed in that commit. For method names we go through the list of classes that were modified in that commit (computed while assessing change-proneness) and determine the class that contains the methods of interest. If a commit does not have any Jira issue keys or specific keywords then it does not represent a bug-fix; therefore, it is not included in our analysis on defect-proneness.

After we identify all the commits in which bugs were repaired, we start our evaluation of the impact of static constructs on change- and defect-proneness. First, we iterate over the commits and determine what was changed between 2 consecutive versions. The basic features provided by Git *dif* were considered insufficient for a thorough analysis, therefore we use a specialized tool called *ChangeDistiller* [96] to extract fine-grained source code changes. The categories of changes that can be identified along with the specific modifications from each category are summarized in Table 3.2. We included all types of changes in our study, even those related to comments and documentation. These were not disregarded because, although they do not represent significant modifications (such as adding new features or fixing bugs), they do ease the understandability of the respective parts of the code. A class might be less susceptible to change due to the fact that its functionalities are understood properly.

Once all the fine-grained changes have been extracted, we can begin our assessment on class change- / defect-proneness. The analyses are similar, but in the one related to defect-proneness we only consider the commits that were categorized as bug-fixes. For each type of static constructs (and their clients), we want to establish 2 things: 1) if the classes that contain them are modified more frequently compared to other similar classes, and 2) whether or not more changes are performed on them per commit. The first aspect is important because if that is the case, then the respective classes can be considered more change-prone (or error-prone if only the bug-fix commits are studied). The second aspect complements the first one; for example, there might be situations in which a production class was modified in a smaller number of commits, but the amount of changes that were performed in each commit is significant. Such a class should be categorized as having a higher change-proneness than one that was modified in more commits but only 1 or 2 changes occurred per commit. Both aspects will be taken into account during the evaluation; in the particular case in which the values obtained for them when comparing 2 classes are contradictory, we will lean towards the one for which the difference is greater. As an example, if one class was modified in 30 commits with an average of 2.3 changes per commit while another was changed in 10 commits and the corresponding average is 2.6, then the first class is considered more change-prone.

Table 3.2 Categories of changes retrieved by *ChangeDistiller*

Change category	**Change**	**Acronym**
ADDED_CLASS	ADDITIONAL_CLASS	AC
REMOVED_CLASS	REMOVED_CLASS	RC
CLASS_DECLARATION	CLASS_RENAMING	CR
	PARENT_CLASS_CHANGE	PCC
	PARENT_CLASS_DELETE	PCD
	PARENT_CLASS_INSERT	PCI
	PARENT_INTERFACE_CHANGE	PIC
	PARENT_INTERFACE_DELETE	PID
	PARENT_INTERFACE_INSERT	PII
	REMOVED_FUNCTIONALITY	RF
	ADDITIONAL_FUNCTIONALITY	AF
METHOD_DECLARATION	RETURN_TYPE_CHANGE	RTC
	RETURN_TYPE_DELETE	RTD
	RETURN_TYPE_INSERT	RTI
	METHOD_RENAMING	MR
	PARAMETER_DELETE	PD
	PARAMETER_INSERT	PI
	PARAMETER_ORDERING_CHANGE	POC

	PARAMETER_RENAMING	PR
	PARAMETER_TYPE_CHANGE	PTC
ATTRIBUTE_DECLARATION	ATTRIBUTE_RENAMING	AR
	ATTRIBUTE_TYPE_CHANGE	ATC
	ADDING_ATTRIBUTE_MODIFIABILITY	AAM
	REMOVING_ATTRIBUTE_MODIFIABILITY	RAM
	ADDITIONAL_OBJECT_STATE	AOS
	REMOVED_OBJECT_STATE	ROS
BODY_STATEMENTS	STATEMENT_DELETE	SD
	STATEMENT_INSERT	SI
	STATEMENT_ORDERING_CHANGE	SOC
	STATEMENT_PARENT_CHANGE	SPC
	STATEMENT_UPDATE	SU
BODY_CONDITIONS	CONDITION_EXPRESSION_CHANGE	CEC
	ALTERNATIVE_PART_DELETE	APD
	ALTERNATIVE_PART_INSERT	API
COMMENTS	COMMENT_DELETE	CD
	COMMENT_INSERT	CI
	COMMENT_MOVE	CM
	COMMENT_UPDATE	CU
DOCUMENTATION	DOC_DELETE	DD
	DOC_INSERT	DI
	DOC_UPDATE	DU
OTHERS	UNCLASSIFIED_CHANGE	UC
	DECREASING_ACCESSIBILITY_CHANGE	DAC
	INCREASING_ACCESSIBILITY_CHANGE	IAC
	ADDING_CLASS_DERIVABILITY	ACD
	ADDING_METHOD_OVERRIDABILITY	AMO
	REMOVING_CLASS_DERIVABILITY	RCD
	REMOVING_METHOD_OVERRIDABILITY	RMO

3.5. Implementation

The process utilized to collect the necessary data is summarized in Figure 3.4 and consists of 3 steps. First we address the presence / usage and the evolution of static constructs (and their clients), as shown in Figure 3.4(a). An Eclipse plugin called *Patrools* [95] was 1) used to extract data related to a system's class structure and 2) extended by us with the proposed detection rules and other measurements that were needed. As an example, for singletons we added a rule that checks if a class does not have any public constructors. Once all the detection strategies have been implemented, the analysis on static construct presence / usage can be conducted. For the latest version of a system we determine the number of instances of each type (as per the categorization from Section 3.1). In terms of static construct usage, we identify the client classes of an instance based on the FAN-IN of the production class that contains it.

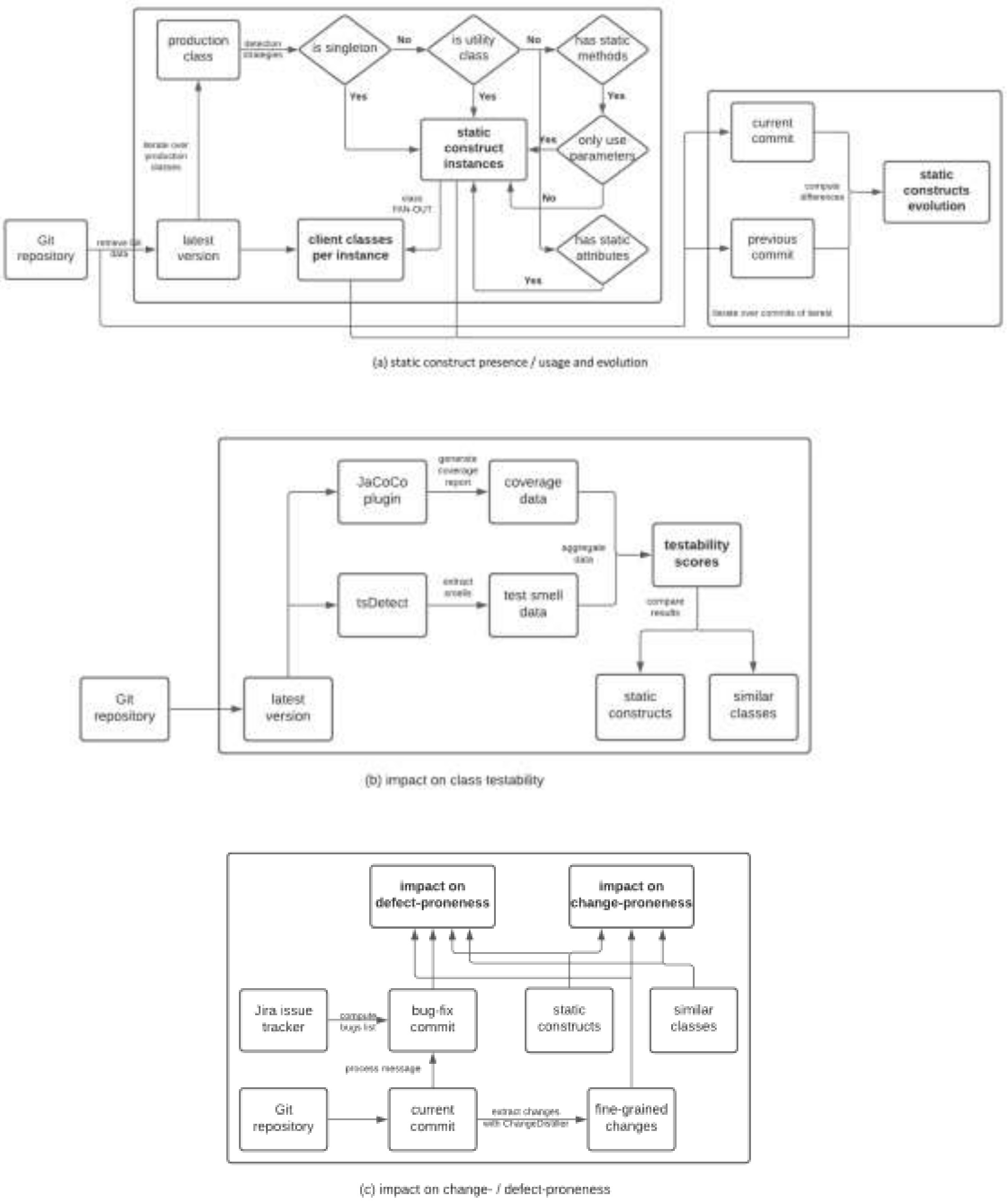

Figure 3.4. Implementation of the data collection process

We also integrated the *jGit* API into *Patrools* and utilized it to retrieve the source code of a system from the corresponding Git repository. We iterate over the commits starting from the initial one and select the last commit of each month as the version of interest. We run the detection strategies described in Section 3.1 in order to identify instances of static constructs together with all the classes that use them and compare the results with those obtained for the commit of interest from the previous month. By doing this, we are able to determine 1) which static constructs were added / removed since the previously studied commit and 2) how many production classes are currently using a particular instance. Other useful information, such as classes being marked as Deprecated, are also recorded and considered in our analysis.

Figure 3.4(b) depicts the process through which the testability of a class is assessed in relation to other similar classes. For the latest version of a project we first build the entire codebase using either *Maven* or *Gradle* (depending on how the system is structured). Then *JaCoCo* is utilized to collect code coverage data, including line and branch coverage, along with an assessment of class / method complexity. The plugin generates a coverage report that is parsed using the *jDom* API in order to extract the values of interest. Besides coverage information, we have also included test smell data in our quantification of class testability. *Patrools* is used to determine all the production classes that are covered by a test class based on the latter's FAN-OUT. A CSV file is created which contains these associations; the file is then provided as input to *tsDetect* [94]. The tool verifies which of the smells are present in each test class and generates another CSV file with the results that is parsed using *OpenCSV*. We extended *tsDetect* so that it reports the smells per unit test, not for an entire test class; this allows for a more thorough analysis on the quality of the unit tests.

Patrools is also utilized to calculate size and complexity measurements for a system's production classes. This is done in order to find classes that are similar to those with the static construct instances and their clients. Similarity is computed using both the complexity scores provided by *JaCoCo* and the *Patrools* measurements mentioned before. Two classes that are identified as similar can then be compared based on their corresponding testability scores. These values are obtained by aggregating the 1) code coverage and 2) test quality measurements described above.

Finally, Figure 3.4(c) illustrates how change- and defect-proneness can be evaluated. First, we establish a HTTP connection to the Jira server for the respective system using the Java *HttpURLConnection* class. Afterwards, subsequent GET requests are performed until all the information related to the issues is collected. The data is retrieved in JSON format and parsed using the *JSON.simple* library. Once this is complete, we rely again on the *jGit* API to fetch the system's source code from the Git repository and iterate over its commits. For each commit we apply text processing techniques on the commit message to extract the information necessary for establishing whether or not it is a bug-fix.

We also compute the differences between each commit and the one before it in order to determine the classes that were modified. As discussed in the previous section, a simple Git *diff* would not have been enough for a thorough analysis, therefore we use ChangeDistiller to obtain fine-grained source code changes. With this tool we are able to gather data related to the production classes that suffered modifications, the entities that were altered, and the changes that were performed. After collecting all the information, we can assess if static constructs and their clients were changed more frequently during normal / bug-fix commits and determine the specific types of the modifications that occurred.

In summary, this chapter discusses:

1. The categorization of static constructs and the detection strategies for each of the following categories:
   - singletons (both stateful and stateless), general form along with several variations of the pattern;
   - utility classes;
   - static methods that access state / only operate on parameters;
   - static non-final attributes and constants;
   - static initialization blocks.
2. The process for studying the evolution of different types of static constructs which includes:
   - the total number of instances from a category;
   - the number of client classes for each instance;
   - additional information (e.g., an instance being marked as Deprecated).
3. The model for evaluating class testability; more specifically, the testability score which is an aggregate of:
   - a quantitative score based on line coverage and the percentage of methods that are covered by unit tests;
   - a qualitative score based on the percentage of tests that contain smells and the number of different types of smells present in a test class.
4. The method of assessing change- / error-proneness which entails:
   - establishing whether or not a commit is a bug-fix;
   - extracting fine-grained source code changes for all the commits of a system;
   - determining if the classes that have static constructs were modified more frequently during normal / bug-fix commits and whether or not more changes were performed on them per commit.
5. The implementation of the entire data collection process.

4. DESIGN OF THE EMPIRICAL STUDY

This chapter explains how we designed the empirical study that was conducted. We start by discussing the main goal of the study and the hypotheses that were formulated. Then we describe the independent and dependent variables considered, together with the procedures through which they were measured. The criteria based on which we selected the systems included in the study are also covered. Finally, the last section of this chapter presents all the analyses that were conducted as part of the empirical study.

4.1. Main goal

As explained in Chapter 1, the goal of this thesis is to provide a better understanding of static constructs, their evolution, and the software quality aspects on which they have a negative impact. In order to achieve it, we formulated 3 research questions:

- RQ1. Are static constructs used in complex software systems?
- RQ2. How have static constructs evolved throughout the lifespan of a project?
- RQ3. Do static constructs have a negative impact on software quality aspects?

The main objective of the empirical study is to obtain answers to these research questions. To do this, we analyse each of these aspects in isolation. First, we investigate what types of static constructs are present in complex software systems and whether or not they are utilized by other production classes. Then we study the evolution of each category of static constructs throughout the lifespan of a system. The effects of using these constructs on several software quality aspects are also considered. By performing these analyses, we will understand which types of static constructs are problematic and should be avoided, thereby aiding developers in creating better systems.

4.2. Formulated hypotheses

As discussed in Section 1.2, we made several assumptions with regard to static constructs, their evolution, and the effects they cause on various software quality aspects. The first major assumption was that instances of such constructs are present in the production code and there are other classes that utilize them. If this assumption does not hold, then there is no reason to proceed with our study. The second assumption addresses the evolution of static constructs; we want to determine if the number of instances increases as a system grows in size. If this is not the case, then we could consider it a first sign that some static constructs are dangerous and the developers have already become aware of the potential problems they cause. Finally, the third assumption is that static constructs have a negative impact on the 3 quality aspects we are investigating. While we do not expect all types of static constructs to be detrimental, we are confident that at least some of them are (e.g., stateful singletons).

In order to establish whether or not these assumptions are true, we formulated a series of hypotheses corresponding to each of them. For every hypothesis we provide a null and an alternative variant; we want to determine which of the variants holds true.

Hypothesis 1

- **Null hypothesis ($H1_{null}$)**: Static constructs rarely appear in complex software systems.
- **Alternative hypothesis ($H1_{alt}$)**: Static constructs are present in the production code and there are other classes that utilize them.

Hypothesis 2

- **Null hypothesis ($H2_{null}$)**: Static constructs are being used less in later iterations of a project compared to the initial versions.
- **Alternative hypothesis ($H2_{alt}$)**: The number of static constructs increases as a system grows in size.

Hypothesis 3

- **Null hypothesis ($H3_{null}$)**: Static constructs do not have a negative impact on software quality.
- **Alternative hypothesis ($H3_{alt}$)**: There are some types of static constructs that negatively affect the software quality aspects that were investigated.

The first hypothesis covers **RQ1**, while the second one addresses **RQ2**. The last hypothesis was refined for each quality aspect of interest. As an example, for testability the null variant would be "Static constructs do not have a negative impact on class testability", while the alternative one is "There are some types of static constructs that negatively affect the testability of the production classes in which they are present / that utilize them". Establishing which of the variants is true for each of these hypotheses represents the main focus of the empirical study. The following sections describe the experiments that were performed in order to validate these hypotheses.

4.3. Independent and dependent variables

We determined the independent and dependent variables for each of the hypotheses and developed methods for measuring them. Table 4.1 provides an overview of these variables along with their measurement procedures. Most of the procedures have already been discussed in the chapter regarding the approach; those that were not are explained in the subsections covering their corresponding hypothesis.

Table 4.1: Independent and dependent variables per hypothesis

Hypothesis	**Independent Variables**	**Procedure**	**Dependent Variables**	**Procedure**
H1	System characteristics	Subsection 4.3.1	Types of static constructs present / utilized	Section 3.1
H2	System size and complexity	Section 3.2	Number of static construct instances / client classes	Section 3.2
H3	Static construct presence / usage	Section 3.1	Impact on the 3 software quality aspects	Sections 3.3 and 3.4

4.3.1. Hypothesis 1

For the first hypothesis, the independent variables are the specific characteristics of a software system. We want to determine if they have an impact on the dependent variables, namely the types of static constructs that appear / are utilized by other classes in the production code. We expect different categories of static constructs to be encountered more frequently depending on the particular characteristics of a project. For example, libraries should have more utility classes compared to other types of systems.

The static construct instances are categorized based on the procedure discussed in Section 3.1. Both the instances and their client classes are identified through the detection strategies introduced in the respective section. The procedure computes the number of instances / clients from each category for the latest version of a project. We are keen to observe which categories appear / are utilized more depending on a system's characteristics. Besides the general characteristics, such as size and complexity, there are several others that will be investigated (e.g., key functionalities). As an example, considering their nature, we expect libraries to have considerably more static methods than other types of projects.

4.3.2. Hypothesis 2

The second hypothesis addresses the evolution of static constructs and their usage. It tries to establish whether or not more static constructs are introduced as a system grows in size and becomes more complex. While additional instances should appear as new classes are created for the respective project, this might not necessarily be the case; if some types of static constructs have been proven harmful to one or more software quality aspects, then the developers might refrain themselves from using them in the future. Therefore, of particular interest are instances 1) for which the number of client classes has decreased or 2) that were completely removed from the production code.

The independent variables for this hypothesis are the metrics related to size and complexity for a particular version of the project. While the general trend is that more classes are added and the existing ones become increasingly more complex as a system evolves, there might be some versions (e.g., refactorings) in which the number of classes / methods or their complexity decreases. We want to see what happens with the number of instances of static constructs and the classes that utilize them especially when such situations occur. Also, we will try to go beyond just the numbers and understand the reasons why a static construct was removed or lost a considerable amount of clients. The measurement procedures for both the dependent and the independent variables were discussed extensively in Section 3.2.

4.3.3. Hypothesis 3

For the final hypothesis, the independent variables are the different categories of static construct instances and their client classes. We want to establish whether or not they have a negative impact on the 3 software quality aspects that are investigated, namely class testability, change-proneness, and defect-proneness. While some types (such as constants) should not make testing more difficult, there are others (e.g., stateful singletons or static non-final attributes) that might be extremely harmful. We will investigate every category of static constructs in isolation for each quality aspect.

As previously mentioned, the process for categorizing and detecting the instances (and their clients) is described in Section 3.1. Sections 3.3 and 3.4 contain the methods through which we quantify the 3 quality aspects. Class testability is evaluated both from a qualitative and a quantitative perspective; this assessment enables us to obtain a testability score for each production class. Change- and defect-proneness are determined in a similar manner, the only difference being that for the latter only bug-fix commits are taken into account (not the entire commit history). A class is considered change- / error-prone if it was modified more frequently throughout its existence and more changes were performed on it compared to other similar production classes.

4.4. System selection

When choosing the systems for the empirical study we took into account a number of criteria, including the ones discussed by Pinto *et al.* in [97]. The projects needed to be:

- relevant in terms of size and complexity (especially the production code). We tried to avoid trivial systems as they do not represent appropriate examples. Therefore, we selected projects that have a large number of classes / methods and complex hierarchies. The smallest system (Digester) has roughly 200 classes, while others have up to 2000 (e.g., Tomcat).
- available through Git and have a substantial number of versions. Considering that we are studying the evolution of different types of static constructs, the systems need to have a corresponding Git repository that contains their commit history. We selected projects with a large amount of commits; the number of versions ranges from 800 (jHotDraw) to more than 23000 (Tomcat). Additionally, all the systems are still in active maintenance; there are no projects that have not received an update in more than several months.
- extensively covered by unit tests (both line and branch coverage were considered). Class testability is one of the three software quality aspects investigated in our study. We evaluate it based on the quantity and the quality of the associated unit tests; thus it would not make sense to include projects that do not have an appropriate test suite. The only exception is jHotDraw, a system which is not tested properly; we decided to include this project because it is used as reference in [91], an article which addresses the different variations of the Singleton anti-pattern (that we are also detecting). Besides this project, the ratio between the number

of lines of test and production code for the rest of the systems is at least 0.5.

- associated with a Jira issue tracker. To study defect-proneness we need to identify the commits in which bugs were fixed. As explained in Section 3.4, if the commit message contains a Jira key, then we rely on the corresponding issue tracker to determine whether or not the respective key is related to a bug. Therefore, the selected projects should have a Jira instance with all the issues that were created during development available for analysis.

Table 4.2 Overview of the selected systems

System	**# Versions**	**First version**				**Last version**				**Test / code ratio**
		# Classes	**# Methods**	**# Tests**	**Release date**	**# Classes**	**# Methods**	**# Tests**	**Release date**	
BCEL	1704	359	2897	0	29/09/2001	432	3749	383	19/04/2021	0.99
Commons Collections	3567	6	113	64	14/03/2001	525	4451	3523	19/04/2021	1.44
Commons Lang	6330	14	199	294	19/06/2002	318	3599	4567	19/04/2021	1.76
Commons Math	6622	4	85	22	12/04/2003	820	5800	5471	13/04/2021	0.69
Digester	2187	14	176	9	22/04/2001	188	927	768	19/04/2021	0.70
Geode	10173	4992	56289	29812	29/03/2015	4528	55799	24775	20/04/2021	0.52
jHotDraw	804	1	6	0	12/09/2000	291	2713	200	22/05/2020	0.05
Pig	3696	177	932	177	29/09/2007	1756	11870	5706	15/10/2020	0.59
Spring Core	22423	167	1059	608	21/09/2008	646	4827	4124	21/04/2021	0.54
Tomcat	23127	1024	10771	0	27/02/2006	2126	21180	6637	20/04/2021	0.53
Wicket	21060	188	1068	502	01/09/2004	1235	8094	5217	21/04/2021	0.56

Based on the above criteria, we selected 11 projects to be included in the empirical study. We tried to choose systems that differ in terms of 1) size and complexity, 2) development practices, and 3) testing effort, while still meeting the criteria. Table 4.2 presents an overview of the main characteristics of the chosen projects; it shows the number of versions studied (column 2), metrics gathered for the first version of a system and the last release considered (columns 3-10), and the test / production code ratio for the latest version (column 11). A visual representation of the collected metrics is provided in Figures 4.1-4.3. Geode was not included in these visualizations because of its considerably higher values compared to the other projects, which would make the rest of the data more difficult to interpret.

The initial version considered is the first commit in which actual code was present (not just configuration files and documentation). Besides the initial and final versions of a project, we also included intermediate ones when trying to illustrate how the 11 systems have grown in terms of number of classes, methods, and unit tests. These intermediate versions are the last commits of each year for the entire lifespan of a project. A general observation would be that the number of classes / methods increased considerably in the first years of development, and then they remained constant or even decreased (e.g., Commons Math) once a system reached maturity. This is an important consideration that should be kept in mind when studying the evolution of static constructs. The number of unit tests follows a similar evolutionary pattern (especially for the systems that are extensively tested). However, there are several cases in which no unit tests were present in the initial version; they were added in subsequent commits.

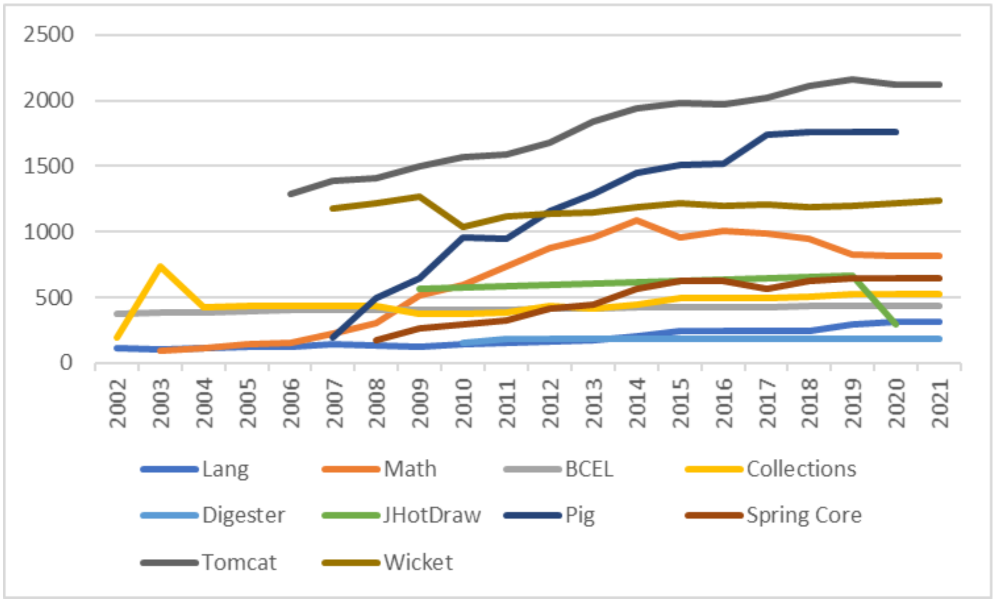

Figure 4.1. Evolution of the number of classes for each project

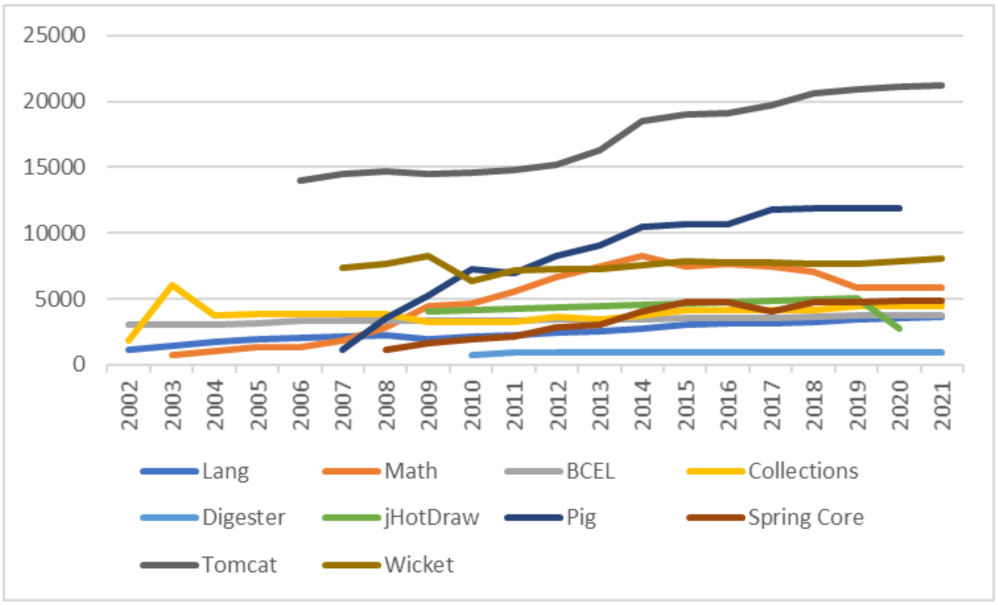

Figure 4.2. Evolution of the number of methods for each project

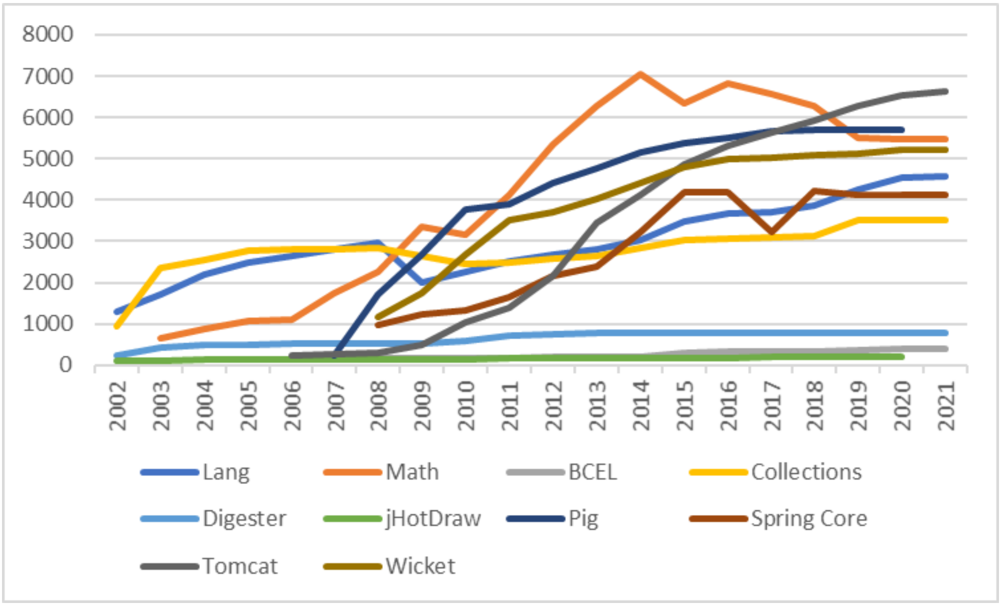

Figure 4.3. Evolution of the number of unit tests for each project

4.5. Analyses conducted

We begin the empirical study with a preliminary analysis of the selected systems; this allows for a better understanding of 1) a system's size and structure, 2) its history, and 3) the quantity and quality of the testing that was performed on its latest version. The following 3 analyses are directly related to the research questions that were formulated. The first one addresses the static construct instances present in the production code, their types and the other classes that utilize them (their clients). In the second analysis we study how instances from each category have evolved throughout a project's lifespan. Finally, the last analysis evaluates the impact of each type of static constructs on the 3 software quality aspects considered.

4.5.1. Preliminary analysis

This analysis goes beyond the initial measurements that were performed on the selected systems (which were presented in the previous section). Besides the number of classes and methods, we are also interested in the overall complexity of a system and the class hierarchy. Several other characteristics (such as key functionalities) are also recorded. All this information is extremely important considering that for the first hypothesis the independent variables are the specific characteristics of a project.

With regard to evolution, in addition to the number of versions we also want to determine 1) the average number of classes that were modified during a commit and 2) the average number of fine-grained source code changes that were performed. This allows us to have an idea of the general patterns by which a system evolves. It can serve as a basis for studying if classes that have / utilize static constructs evolve differently, a topic which will be addressed in a following analysis. During this preliminary analysis we also establish which of the commits are bug-fixes (by following the procedure from Section 3.4).

Last but not least, we evaluate the overall quantity and quality of the unit tests for the latest version of a project. For quantity we first perform code coverage measurements; however, we also want to calculate the percentage of production methods that are addressed by at least 1 test. In terms of quality, we are keen to observe 1) which types of smells are present in the test code (based on the categorization from Section 3.3) and 2) the average number of smells per test class. These metrics will allow for an initial assessment of the testing that was performed for the respective system.

4.5.2. Static construct presence / usage

As mentioned above, the main analysis is split into 3 parts. First we study the latest version of a system in order to establish 1) if static constructs are present and 2) how they are utilized. Besides the number of instances, we are also interested in their types (as categorized in Section 3.1). Each category of constructs is analysed in great detail. As an example, for singletons we distinguish between stateful and stateless ones; moreover, the analysis is further refined so that all the singleton variations discussed in [91] are considered.

In terms of usage, we want to go beyond the number of client classes and understand if they are localized in several packages or spread throughout the source code. We compare the results with those obtained for similar classes, thus allowing us to determine if certain types of static constructs are used differently. As discussed in Chapter 3, no relevant results would have been obtained if we compared the usage of a large class which contains several static constructs to that of a trivial class.

4.5.3. Evolution of static constructs

Regarding the evolution of the instances from each category, we analyse monthly commits to establish whether or not such instances / classes that utilize them were added / removed within this timeframe. We are keen to observe if the number of static constructs increased as a system grew in size. Similar to the previous analysis, the instances from each category are studied separately. Special attention is dedicated to cases in which an instance was deleted or marked as Deprecated because we want to understand the reasons behind such a decision.

The number of client classes is also examined for each instance from its creation up to the latest version considered (or until it was removed) and compared to that of similar classes. We are very interested in cases in which the number of clients suddenly dropped and want to see what happened with the respective static construct in previous commits. In our analysis on evolution we use graphs to display how each category of instances / their clients have evolved for every system included in the study. By doing this we are able to visualize the entire process and uncover certain patterns that might appear.

4.5.4. Impact on software quality aspects

In the last analysis we are looking for correlations between the usage of static constructs and lower values for the 3 software quality aspects that are investigated. Each category of static constructs is studied in isolation, thereby allowing us to determine which types have the highest negative effect on the respective quality aspects. For example, to study the impact of singletons (both stateless and stateful) on class testability, we compare the testability score of classes that have / utilize such instances with the scores obtained for other similar classes. More specifically, we want to establish if the latter are 1) covered more thoroughly by unit tests and 2) the tests are of a better quality (in terms of test smells). As explained in Section 3.3, the testing effort is quantified based on line coverage and the percentage of production methods that are addressed by tests. For the test code quality, we look at the percentage of unit tests that contain smells and the different categories of smells present in a test class. The process is repeated for all the other categories of static constructs, thus obtaining a better understanding of the impact of each type on testability.

The effect on the other 2 quality aspects is investigated in a similar manner. Both for change- and defect-proneness we try to determine if the classes that contain / utilize different types of static constructs 1) are modified more frequently and 2) more changes are performed on them compared to other production classes. The only difference between the 2 quality aspects is that for error-proneness we only consider the commits that were categorized as bug-fixes (as explained in Section 3.4). This is done for each category of static constructs, thus enabling us to pinpoint which types are the most detrimental. Some types may have a negative impact on only 1 or 2 of the aspects, while others might affect all 3. The latter are the most problematic and should be avoided at all cost.

In summary, this chapter discusses:

1. The main goal of this thesis, namely to answer the 3 research questions that address:
   - static constructs presence / usage;
   - the evolution of different types of static constructs;
   - their impact on 3 software quality aspects: testability, change-proneness, and defect-proneness.
2. The hypotheses that were formulated; for each research question there are 2 hypotheses, a null version and an alternative one.
3. The independent and dependent variables for each hypothesis along with their corresponding measurement procedures.
4. The system selection process with an emphasis on the criteria based on which the projects were chosen:
   - relevancy in terms of size and complexity;
   - availability on Git and a considerable amount of versions;
   - appropriate coverage through unit testing;
   - availability of a corresponding Jira issue tracker.
5. The analyses that were conducted as part of the empirical study:
   - a preliminary analysis on the size and structure of the systems, their history, and the effort that was put into testing them;
   - an analysis on the presence / usage of different types of static constructs;
   - an analysis on the evolution of each of the respective types;
   - 3 analyses on the impact of static constructs on the quality aspects investigated.

-

5. RESULTS

5.1. Static constructs identified

First we studied the different types of static constructs that appear in the latest version of a project. Besides the number of instances, we also wanted to determine how they are utilized and whether or not their clients are localized or spread throughout the system. For attributes we only computed the average number of methods that access them from other production classes; because the values were so low (there are very few such methods), we decided not to calculate the average number of packages from which the attributes are accessed as those values would have been even lower. Also, the initialization blocks are a special kind of static constructs, they do not have any clients nor are there other constructs that can be considered similar to them; therefore, for this category we only calculated the total number of instances. Finally, for static methods we considered the other methods that invoke them as clients (rather than the classes that contain the respective methods); the classes from which they are called were utilized afterwards to study the client spread (instead of packages which were used for singletons / utility classes).

5.1.1. BCEL

Table 5.1.1: Static constructs BCEL

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	20	0.05	-	0.1401	-
	Constants	694	0.2983	-		
Singletons	Stateful	1	65	4	9.3310	2.0886
	Stateless	2	1	1		
Static methods	Utility classes	11	35	2.4545	8.7530	2.0831
	Access state	13	5.1538	2.1538	18.0181	6.1455
	Operate on parameters	119	4.2941	1.5126		
Static init blocks		5	-	-	-	-

There are 1706 attributes in BCEL, 714 (41.85%) of which are static; most of them are constants, only 20 being non-final (1.17%). The constants seem to be used by more classes compared to other non-static attributes (average number of clients of 0.2983 vs. 0.1401), while the non-final ones are not (only 0.05).

From the 432 classes found in the latest version of the system, 3 (0.69%) singletons were identified. The first one, Type, is stateful; however, the 2 non-final attributes are marked as Deprecated and the developers specify that they should be final. The other 2 singletons, DOUBLE_Upper and LONG_Upper, extend the aforementioned class, therefore they were categorized as Subclassed Singletons. The number of clients / packages for the stateful singleton are significantly higher than the averages for the stateless variants or the other production classes, but the results might be skewed due to the fact that this class has several Deprecated attributes and a lot of methods still use them.

There are 11 utility classes (2.55%) that contain a total of 107 methods. Although they have considerably more clients (an average of 35) compared to other classes (8.7530), the package distribution for the client classes is very similar (2.4545 vs. 2.0831). The system has 3749 methods in total, but only 132 (3.52%) are static methods that are not part of singletons or utility classes. From them

13 (0.35%) access their class's state, while 119 (3.17%) only operate on parameters. Both types of methods have fewer clients compared to other methods and their usage is more localized; on average, they are used by roughly 5 methods from 1-2 other classes, while for their non-static counterparts these values are much higher (over 18 methods from more than 6 classes).

Finally, there are 5 classes that contain 1 static initialization blocks each: Utility, ConstantUtf8, InstructionConst, Class2HTML, and InstructionFinder.

5.1.2. Commons Collections

Table 5.1.2: Static constructs Commons Collections

Category		Total # instances	Avg. # clients	Avg. # packages	Avg. # clients similar type	Avg. # pack. similar type
Static attributes	Non-final	0	-	-	0.3044	-
	Constants	260	0.0731	-		
Singletons	Stateful	0	-	-	1.0519	0.7058
	Stateless	5	0.8	0.6		
Static methods	Utility classes	31	3.4194	1.3226	0.9008	0.668
	Access state	1	2	1	15.1232	9.7912
	Operate on parameters	198	2.5909	1.5404		
Static init blocks		0	-	-	-	-

There are 871 attributes in the latest version studied, 260 (29.85%) of which are static. All the static attributes are constants, no static non-final ones were encountered. They have, on average, a lower number of client classes compared to the attributes that are not static (0.0731 vs. 0.3044).

Five of the 525 classes analysed are singletons; they are all stateless and their average number of clients is slightly lower than that of similar classes (0.8 vs. 1.0519). This observation also holds true for the average number of packages from which they are utilized (0.6 vs. 0.7058). In terms of actual types, they are all Eager Instantiations.

There are 4451 methods in the system's production classes. With regard to utility classes, 31 such instances containing 512 methods (11.5%) were found. They have a much higher average number of clients than classes which are similar to them in terms of size and complexity (3.4194 vs. 0.9008). Furthermore, they are utilized from more packages (1.3226 vs. 0.668). This suggests that such classes are a key part of Commons Collections, a project which is structured as a library.

There are only 199 static methods (4.47%) that are not part of singletons or utility classes. One of them accesses its class's state, while the other 198 (4.45%) only operate on parameters. These methods are called, on average, by roughly 2 other methods from 1-2 classes; these values are significantly lower than for non-static methods (15.1232 methods and 9.7912 classes).

No static initialization blocks were found in the version of Collections that was analysed.

5.1.3. Commons Lang

For Commons Lang, which is also structured as a library, there are numerous static attributes and methods. From the 862 attributes present in the system's classes 526 of them are static, which is over 61%. Most of the static attributes represent constants and are generally used only in the class in which they are declared, therefore the average number of clients is very low (0.0913). It is surprising that the corresponding value for non-static attributes is even lower (0.0774), however the difference is not significant. Only 1 of the static attributes is non-final, defaultStyle from the ToStringBuilder class, which has 0 clients. In terms of packages from which the attributes are utilized, it is difficult to make a distinction between the different types of attributes due to the small number of client classes.

Table 5.1.3: Static constructs Commons Lang

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	1	0	-	0.0774	-
	Constants	525	0.0913	-		
Singletons	Stateful	0	-	-	1.4227	0.7828
	Stateless	1	0	0		
Static methods	Utility classes	51	2.6471	1.4847	1.1835	0.6448
	Access state	3	1.3333	0.6667	3.0867	1.6488
	Operate on parameters	204	1.049	0.5931		
Static init blocks		15	-	-	-	-

In terms of singletons, only 1 instance was identified. ObjectToStringComparator is a stateless singleton of type Eager Instantiation; there are no production classes in the latest version of Commons Lang that utilize it.

Similarly to attributes, 1702 out of the 3599 methods are static (47.29%). There are 51 utility classes which contain a total of 1495 methods (41.54%). They have, on average, more clients (2.6471 vs. 1.1835) from more packages (1.4847 vs. 0.6448) compared to other production classes. The rest of the static methods are divided into 2 categories, the ones that access their class's state (3) and those that only operate on parameters (204). It can be observed that these types of static methods have, on average, a lower number of client methods compared to other non-static methods and are utilized from fewer classes (around 0.6 for both types vs. 1.6488).

Finally, there are 15 static initialization blocks in 11 production classes. Most of the classes contain 1 such instance, but there are 2 which contain more (ClassUtils and FieldUtils with 4 and 2 instances, respectively).

5.1.4. Commons Math

Table 5.1.4: Static constructs Commons Math

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	12	0	0	0.1141	-
	Constants	632	0.0934	-		
Singletons	Stateful	0	-	-	3.9048	1.3443
	Stateless	1	1	1		
Static methods	Utility classes	25	17.72	4.4	3.4667	1.2478
	Access state	10	5.9	3.2	27.9794	6.8239
	Operate on parameters	181	4.453	1.8287		
Static init blocks		12	-	-	-	-

There are 644 static attributes (30.31%) from a total of 2125; only 12 (0.56%) are non-final while the other 632 (29.74%) are constants. The non-final ones are only used in the class in which they are declared; the constants are rarely utilized in other production classes, the average number of clients is comparable to that of the non-static attributes (0.0934 vs. 0.1141).

Only 1 singleton was detected in the latest version of the system, Decimal64Field. It is of type Lazy Instantiation and has a single client (Decimal64). From the 820 production classes in the project 25 are utility classes (3.05%), much fewer than the previous 2 libraries. They contain a total of 414 static methods and are used, on average, by 17.72 classes, which is significantly higher than the average number of clients for other classes (3.4667). In terms of localization, the clients are also more spread out throughout the project. From the total of 5800 methods, 10 (0.17%) are static and access their class's state while 181 (3.12%) operate only on parameters. These categories of methods have a smaller number of clients (on average 5.9 and 4.453, respectively) compared to other non-static methods (27.9794). Correspondingly, the average number of classes from which they are utilized is also lower.

There are 12 static initialization blocks in the system's production classes. Only 1 class contains more than 1 initialization block, FashMath which has 3 such instances.

5.1.5. Digester

Table 5.1.5: Static constructs Digester

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	0	-	-	0.0311	-
	Constants	36	0	-		
Singletons	Stateful	0	-	-	-	-
	Stateless	0	-	-		
Static methods	Utility classes	2	1.5	1	2.6882	1.3011
	Access state	0	-	-	4.1058	1.9593
	Operate on parameters	7	2	2		
Static init blocks		0	-	-	-	-

For the smallest system that was analysed, several types of static constructs are missing. From the 325 attributes found in the latest version of Digester, only 36 are static (which is roughly 10%). All of them are constants, there are no static non-final attributes. The constants do not have any clients, they are utilized only in the classes that declare them. The average number of classes from which the non-static attributes are used is also very low, thus suggesting that the developers have a strict policy of not accessing attributes from other classes directly.

No singletons were identified for this system, which only has 188 classes. There are however 2 utility classes (1.06%), AnnotationsUtils and LogUtils, that have a total of 8 static methods. They have less clients (1.5 vs. 2.6882) compared to other production classes and are utilized from fewer packages (1 vs. 1.3011). Out of 927 methods, there are no static methods that access their class's state and only 7 static methods that operate on parameters (0.76%). Their average number of clients is considerably lower than for non-static methods (2 vs. 4.2355), but the average number of classes from which they are called is roughly the same (2 vs. 1.9593). Just like for singletons, there are no production classes that contain static initialization blocks.

5.1.6. Geode

Table 5.1.6: Static constructs Geode

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	308	0.3019	-	0.2832	-
	Constants	8030	0.2391	-		
Singletons	Stateful	17	7.2611	2.2906	6.8429	2.2489
	Stateless	64	2.5601	0.8804		
Static methods	Utility classes	243	8.5144	2.4198	8.8869	2.9502
	Access state	184	18.8587	10.0271	39.9615	17.3232
	Operate on parameters	1804	20.5937	12.0061		
Static init blocks		107	-	-	-	-

There are 22176 attributes in total in Geode's production classes, 8338 (over 37.5%) of which are static. From the static ones, 8030 (96.31%) of them are constants while the remaining 308 are static non-final. There is no significant difference between the average number of clients for the static non-final attributes (0.3019) compared to constants (0.2391). The values obtained are also similar to the one for the non-static attributes (0.2832).

From a total of 4992 production classes, 81 are singletons (1.62%); 17 of them are stateful and 64 stateless. Subclassed Singleton appears to be the predominant type as 51 singletons are children of BaseCommand and 4 are of InternalFunction. The rest of the singletons are either Lazy Instantiations or Eager Instantiations, the sole exception being HexThreadIdPatternConverter which is a Limiton. Unlike the other systems, Geode has 3 singletons that are marked as Deprecated in its latest version. In terms of usage, the average number of clients / packages from which stateful singletons are utilized is comparable to that of similar classes (7.2611 vs. 6.8429 and 2.2906 vs. 2.2489, respectively). However, the corresponding value for the stateless variants are considerably lower (2.5601 for clients and 0.8804 for packages).

The version studied only has 243 utility classes (5.37%) that contain a total of 1391 static methods. Both their average number of clients and the average number of packages from which they are used are resemble the ones obtained for similar classes (8.5144 vs. 8.8869 and 2.4198 vs. 2.9502, respectively); this show that for Geode the usage patterns for utility classes are no different to those of other production classes. Out of a total of 55671 methods, only 3379 are static for this system. From these 184 access their class's state (0.33%) while 1804 solely operate on parameters (3.24%). Both types are invoked by a comparable number of other methods (18.8587 and 20.5937); these averages are lower than the one obtained for the non-static methods (39.9615). Consequently, the average number of classes from which they are called is also smaller (10.0271 and 12.0061 vs. 17.3232).

Only 107 static initialization blocks were found in the system's production classes. Most of the classes have 1 such instance, but there are some that have more; the ones with more than 2 static initialization blocks are: NativeCallsJNAImpl (5), LinuxNativeCalls (4), and FreeListManager (3).

5.1.7. jHotDraw

Table 5.1.7: Static constructs jHotDraw

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	25	0.04	-	0.1992	-
	Constants	369	0.0705	-		
Singletons	Stateful	0	-	-	4.11	1.3093
	Stateless	1	3	2		
Static methods	Utility classes	4	6.75	2.25	4.0694	1.2986
	Access state	6	0.8333	0.8333	9.8773	4.5476
	Operate on parameters	39	2.9231	2.8462		
Static init blocks		2	-	-	-	-

From the 866 attributes present in the latest version of jHotDraw, 394 are static (which is roughly 45.5%). Only 25 of them are non-final (2.89%), while the other 369 are constants (42.61%). Both types of static attributes have less clients than their non-static counterparts, but all the averages are very low.

In terms of singletons, only 1 stateless instance was found (FigureLayerComparator) which is of the Eager Instantiation type. It has 3 clients that are localized in 2 nested packages; other similar production classes have, on average, more clients (4.11), but they are generally from the same package.

Out of the 292 production classes 4 (1.37%) are utility classes: ResizeHandleKey, AttributeKeys, TransformHandleKit, and PaletteUtilities. They contain a total of 45 static methods, have more clients than other production classes (6.75 vs. 4.0694), and these clients are more spread out through the code (2.25 vs. 1.2986 packages on average). There are 2719 methods in the version analysed, but only 45 (1.66%) of them are static and not part of singletons or utility classes. Six (0.22%) access state, while the other 39 (1.43%) only operate on their parameters. These static methods have fewer clients than their non-static counterparts (especially the ones from the first category) and their utilization is more localized (0.8333 classes the former and 2.8462 the latter vs. 4.5476 for non-static methods).

There are only 2 classes that contain 1 static initialization block each, DefaultDrawingView and AttributeKeys.

5.1.8. Pig

Table 5.1.8: Static constructs Pig

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	245	0.151	-	0.3133	-
	Constants	1101	0.3406	-		
Singletons	Stateful	14	13.9286	4.9286	6.435	2.8546
	Stateless	4	37.5	10.75		
Static methods	Utility classes	75	11.2667	4.2133	6.356	2.5172
	Access state	86	26.4651	13.4186	42.4463	15.8513
	Operate on parameters	451	9.6186	3.8315		
Static init blocks		34	-	-	-	-

The system's production classes have 4423 attributes, out of which 1346 (30.43%) are static. From these 1101 are constants (24.84%) while the remaining 245 are static non-final (5.55%). The average number of clients for the attributes that are non-final is lower (0.151) than the corresponding values for constants (0.341) or non-static attributes (0.3133) which are comparable.

From the 1756 classes in the latest version studied, 18 of them are singletons (1.03%). Unlike what was observed thus far, Pig is a project in which most of the singletons (14) are stateful. The stateful singletons have more than double (13.9286) the number of clients when compared to the average obtained for similar classes (6.435). However, the most surprising finding would be the average number of clients for the 4 stateless singletons (37.5); this is mainly due to 1 class, TupleFactory, having a large amount of clients (128). The number of packages from which singletons are utilized is also considerably higher (4.9286 for stateful and 10.75 for stateless singletons vs. 2.8546 for similar classes).

There are 75 utility classes (4.27%) containing 520 static methods. They have, on average, 11.2667 clients which is almost double than the corresponding value for similar classes (6.356). Furthermore, the average number of packages from which utility classes are called is also higher (4.2133 vs. 2.5172). From the 9050 methods found, only 86 (0.95%) are static ones that access state while 451 (4.98%) are static and solely operate on their parameters. They also have fewer clients, 26.4651 for the first category and 9.6186 for the latter, thus suggesting that they are not called as frequently in projects such as Pig.

Finally, there are 34 static initialization blocks in 31 of the system's classes. Most of the classes contain 1 such block, but there are 3 classes that contain 2, TezJobSplitWriter, JrubyScriptEngine and Main.

5.1.9. Spring Core

Roughly a third (621) of the project's 1911 attributes are static. Most of them (611) are constants, while the remaining 10 are static non-final. The average number of clients for the non-final attributes is lower (0.3) than the values obtained for constants (0.5827) and non-static ones (0.6).

Table 5.1.9: Static constructs Spring Core

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	10	0.3	-	0.6	-
	Constants	611	0.5827	-		
Singletons	Stateful	0	-	-	3.3239	1.0845
	Stateless	5	2	0.8		
Static methods	Utility classes	63	7.5397	2.1111	2.9177	0.9811
	Access state	13	3.6923	2.2308	5.2999	2.7307
	Operate on parameters	236	3.6017	2.0593		
Static init blocks		27	-	-	-	-

Five classes (0.77%) were categorized as singletons from the system’s 646 classes. All the singletons are of type Eager Instantiation and have on average 2 clients, which is lower than the average number of clients for other similar classes (3.3239). The average number of packages from which they are accessed is also slightly lower (0.8 vs. 1.0845).

There are 63 utility classes which is almost 10% of the Spring Core’s production classes. They contain 768 static methods and have considerably more clients (7.5397 on average) than the rest of the classes (2.9177); their clients are also more spread out, they are part of 2 or more packages while the ones for similar classes are either in the same package or in 1 more. There are 4827 methods in total for the latest version of the project, but only 13 (0.27%) access state and 236 (4.89%) operate on parameters. They have a lower number of clients (3.6923 and 3.6017, respectively) compared to the non-static methods (5.2999); the average number of packages from which the static methods are called is also smaller, albeit not by much (2.2308 and 2.0593 vs. 2.7307).

The latest version of Spring Core studied contains 27 static initialization blocks. There are only 2 production classes with more than 1 such block, ReflectUtils and ReflectionUtils (both with 2).

5.1.10. Tomcat

Table 5.1.10: Static constructs Tomcat

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	134	0.1119	-	0.1344	-
	Constants	3106	0.1806	-		
Singletons	Stateful	8	1.5	0.625	1.5724	0.6555
	Stateless	5	3.2	1.4		
Static methods	Utility classes	149	0.8322	0.4966	1.2004	0.5105
	Access state	46	3.1304	1.8478	5.7686	2.689
	Operate on parameters	420	3.669	1.1024		
Static init blocks		99	-	-	-	-

From a total of 9652 attributes there are 134 static non-final ones (1.39%) and 3106 constants (32.18%). The average number of clients is very small for both types (0.1119 for the former and 0.1806 for the latter); they are comparable to the value obtained for the non-static attributes (0.1344).

There are 13 singletons (0.61%) from the system’s 2126 production classes, 8 of which are stateful; Tomcat in only the second project for which there are more stateful variants than stateless ones. There is little difference between the average number of clients / packages for stateful singletons and other classes that are similar to them in terms of size and complexity (1.5 / 0.625 vs. 1.5724 / 0.6555). For stateless singletons the corresponding value are significantly higher (3.2 for clients and 1.4 for packages).

The number of utility classes is also quite low; there are only 149 instances (around 7%). They are utilized less compared to singletons or similar classes; the average number of clients is 0.8322, while for packages it is 0.4966. Similar observations can be made with regard to other static methods; there are only 46 that access state and 420 which solely operate on parameters. All of these values are low considering the size of Tomcat. The average number of methods that invoke these instances is also lower than the one obtained for non-static methods (3.1304 and 3.669 vs. 5.7686); furthermore, they are called from fewer classes (1.8478 and 1.1024 vs. 2.689).

Finally, there are 99 instances of static initialization blocks in 95 of Tomcat's production classes; there are 4 classes that contain 2 such instances.

5.1.11. Wicket

Table 5.1.11: Static constructs Wicket

Category		**Total # instances**	**Avg. # clients**	**Avg. # packages**	**Avg. # clients similar type**	**Avg. # pack. similar type**
Static attributes	Non-final	35	0.0857	-	0.1059	-
	Constants	1192	0.0998	-		
Singletons	Stateful	0	-	-	3.8	1.8973
	Stateless	8	4	2.125		
Static methods	Utility classes	31	3.6774	2.6452	3.8032	1.8851
	Access state	6	49.1667	25.1667	15.1127	7.4537
	Operate on parameters	209	22.8995	24.5694		
Static init blocks		3	-	-	-	-

There are 3011 attributes in all of Wicket's classes, 1227 of which are static (roughly 40%); 35 of the static ones are non-final, while the vast majority are constants. The average number of clients is very similar for all 3 types of attributes (static non-final, constants, and non-static); it can be observed that they are rarely utilized in other production classes.

From the 1423 classes present in the latest version of the project, 9 were categorized as singletons (0.63%). All the singletons are stateless and of type Eager Instantiation. One of the instances, EmailAddressPatternValidator, is marked as Deprecated in the version studied. In terms of client classes, there seems to be very little difference between the average number of clients / packages from which stateful singletons are utilized compared to other production classes (4 vs. 3.8 / 2.125 vs. 1.8973).

There are 31 utility classes (2.18%) that contain a total of 172 static methods. The observation that was made with regard to the average number of clients / packages for singletons also applies for utility classes, although their usage is a little bit more spread out (2.6452 vs. 1.8851 packages). Out of the 9162 methods found, only 6 are static and access state (0.07%) while 209 (2.28%) solely operate on parameters. Both types are called by more methods (49.1667 and 22.8995 vs. 15.1127) than their non-static counterparts; the methods that invoke them are also part of more classes (roughly 25 in both cases compared to 7.4537).

Only 3 classes have static initialization blocks, WicketTagIdentifier (2 such instances), TagUtils, and JavaSerializer (both with 1 instance).

5.2. Evolution of static constructs

Next, we studied the evolution of different types of static constructs. We did not focus solely on the number of instances present in a specific version of the system; we also wanted to establish the reasoning behind the addition / removal of certain instances or their clients. Furthermore, for the class-level constructs (singletons and utility classes) we also analysed their usage throughout the lifespan of the project. It would have been much more difficult to study this aspect for the more fine-grained static constructs (e.g., static non-final attributes or initialization blocks) due to the reasons discussed in the previous section. For each case we created a graph that depicts the total number of instances / the percentage of production classes that utilize instances of that type (y-axis) over time (x-axis). We try to explain why certain situations occur, such as a large decrease of the instances of interest or a class loosing most of its clients.

5.2.1. BCEL

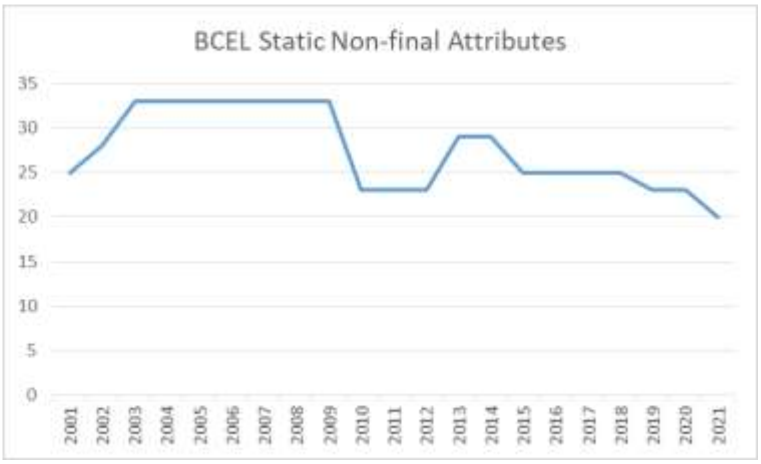

Figure 5.2.1.1: Evolution of static non-final attributes for BCEL

There are very few static non-final attributes compared to constants or non-static ones. We found 25 such instances in the initial version studied. Then this number increased to 33 in May 2003 and remained constant for roughly 6 years even though other types of attributes were being added. From there on it started to decrease with one exception; between May 2013 and August 2015 the number of static non-final attributes increased from 23 to 29, remained constant for about 1 year, and then dropped to 25. Finally, there are only 20 instances in the latest version that was analysed.

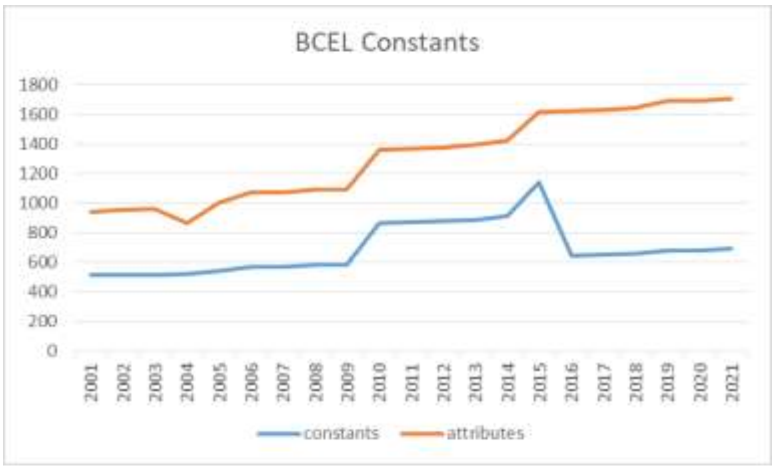

Figure 5.2.1.2: Evolution of constants for BCEL

In the first version of BCEL 514 (54.68%) of the 940 attributes were constants. Up until September 2015 both the number of constants and the total number of attributes increased constantly; at that moment there were 1138 constants (over 70%) out of 1618 attributes. From there on the number of attributes continued to grow while the number of constants dropped to 648. Then it slowly increased; in the latest version of BCEL there are 694 such attributes (40.75%) from a total of 1703.

Four singletons were found in the very first versions of BCEL. Two of them, LONG_Upper and DOUBLE_Upper, are stateless and of type Subclassed Singleton while the others (Type and BranchHandle) are stateful. With the exception of BranchHandle they were part of the project for its entire existence. In August 2002 another stateless singleton was introduced, InstructionComparator; it was of type Eager Instantiation and remained in the system until August 2015. A series of interesting events occurred in the respective time period; for example, BranchHandle became a stateless singleton and remained in that form until June 2016 when state was added to it again. Finally, in February 2019 BranchHandle was removed along with its superclass.

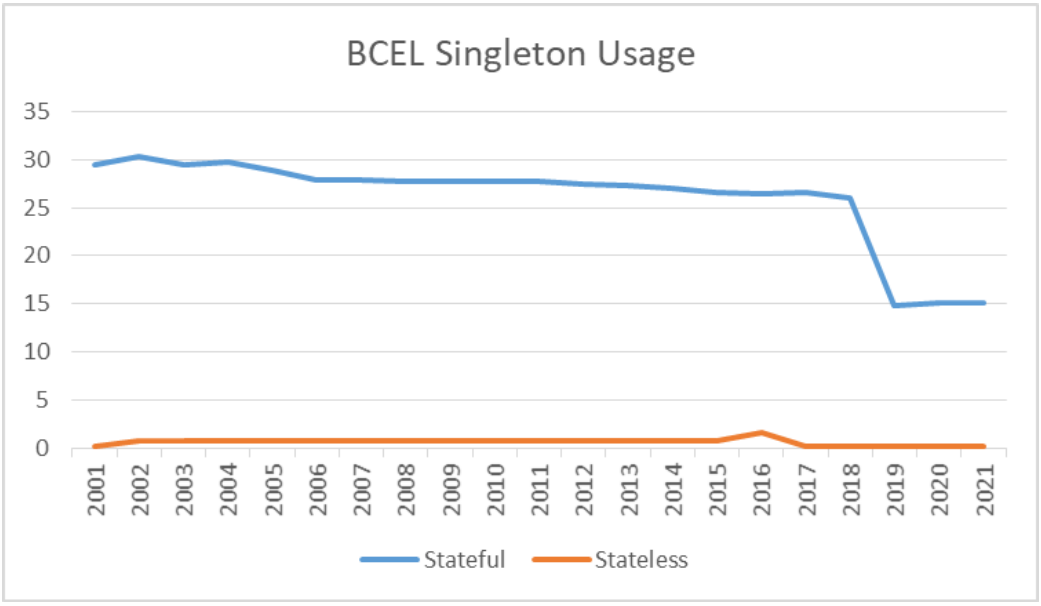

Figure 5.2.1.3: Evolution of singleton usage for BCEL

The usage of stateful singletons is high due to the fact that Type (a class with 61 clients) was implemented as a loose variation of the pattern. Initially, the percentage of production classes that utilize singletons was around 30%; then it started to slowly decrease until reaching 26.04% in 2018. During that time one of the singletons (BrachHandle) was removed, thereby causing the percentage to drop to 14.8%. At the end of the development process a number of production classes were removed; this caused a slight increase in stateful singleton usage (to 15.05%). The percentages for the usage of stateless singletons are very low compared to their stateful counterparts. Between 2002 and 2015 they were roughly around 0.75%. The value increased to 1.67% when BranchHandle became stateless and dropped all the way to 0.23% when state was added back to it.

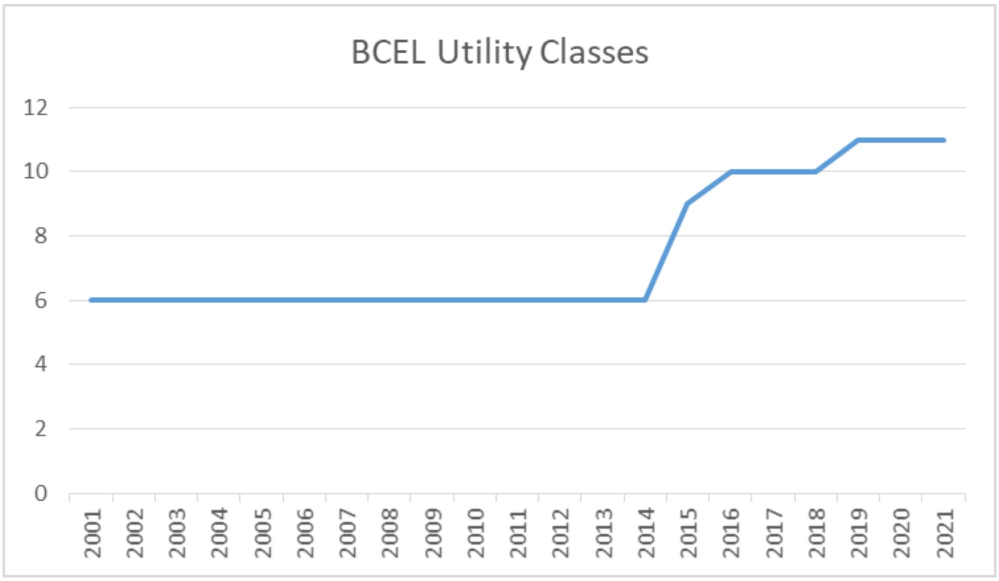

Figure 5.2.1.4: Evolution of utility classes for BCEL

The initial version of the project contained 6 utility classes. The number of instances remained the same until August 2015 when it grew to 9; then it increased again to 10 in May 2016 and to 11 in July 2019. It can be observed that the number of utility classes does not grow constantly as the system increases in size. In terms of usage, while there were only 6 utility classes the percentage of production classes that utilized them was constant (between 10% and 11%). When new instances were added this percentage spiked to 64.82% because one of them (Constants) had 259 clients (from a total of 415 classes). Then the usage increased by a small margin to 66.2% for the latest version studied.

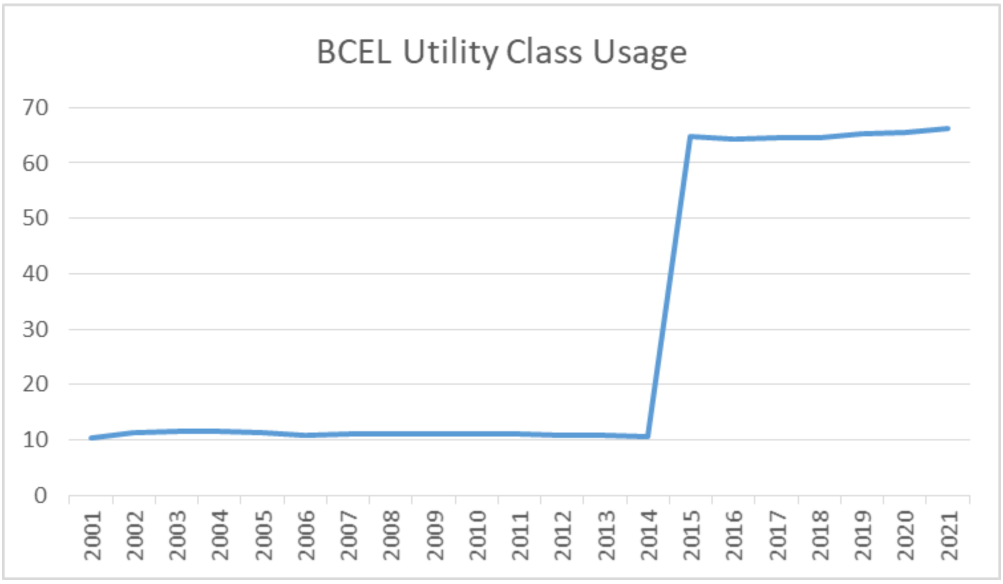

Figure 5.2.1.5: Evolution of utility class usage for BCEL

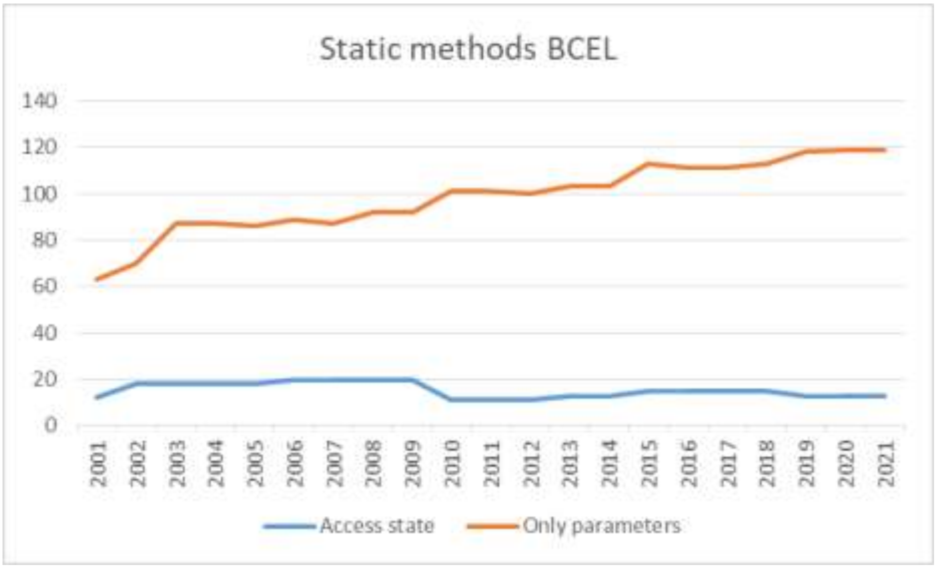

Figure 5.2.1.6: Evolution of static methods for BCEL

There were 12 static methods that access state (0.43%) and 63 that only operate on parameters (2.23) out of a total of 2822 methods. The number of methods from the first category increased slightly in the first years of development until reaching a maximum of 20 in June 2006. It remained constant for almost 4 years, then it dropped to 11 in May 2010. From there on, the number of instances fluctuated; there are 13 static methods that access state (0.35%) in the latest version of the system. On the other have, the amount of static methods that solely operate on parameters increased constantly throughout the years. There were 87 such instances in 2003, 101 in 2010, and 113 in 2018. A maximum value of 119 (3.27%) was reached in April 2020; no such methods were added ever since, but their percentage decreased to 3.17%.

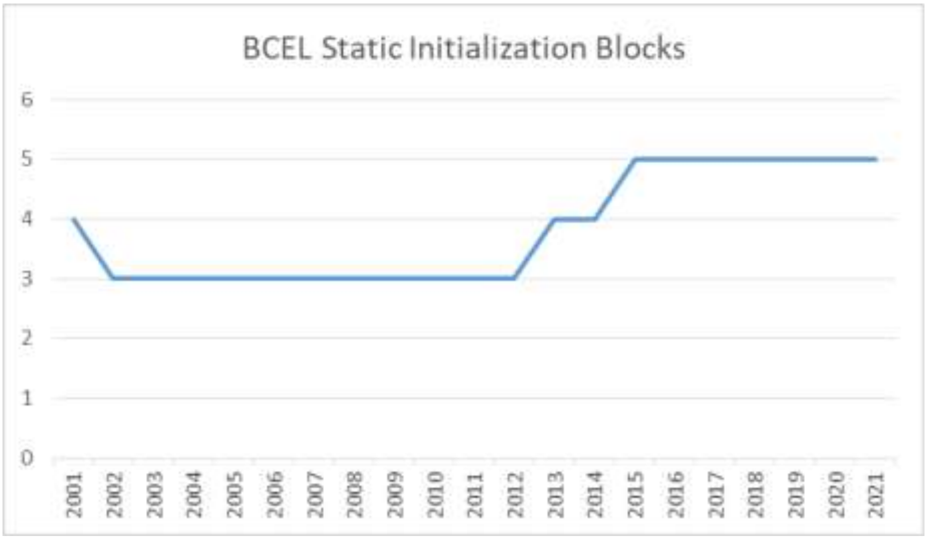

Figure 5.2.1.7: Evolution of static initialization blocks for BCEL

There were 4 static initialization blocks in the first version of BCEL that was studies. However, a class that contained such an instance (Repository) was removed in June 2002. Another instance was added in May 2013 as part of the ConstantUtf8 class. Finally, the fifth and last static initialization block was created in August 2015 in Class2HTML; the number of instances has remained constant ever since.

5.2.2. Commons Collections

There were 1 static non-final attribute and 1 constant in the initial version of the project which contained a total of 35 attributes. The number of static non-final attributes grew in the first year of development until reaching 7 in February 2002. From there it dropped to 3 (even though the number of attributes was continuously growing) and remained constant until June 2018. All 3 instance were removed the following month; there are no static non-final attributes in the latest version of Commons Collections that was studied.

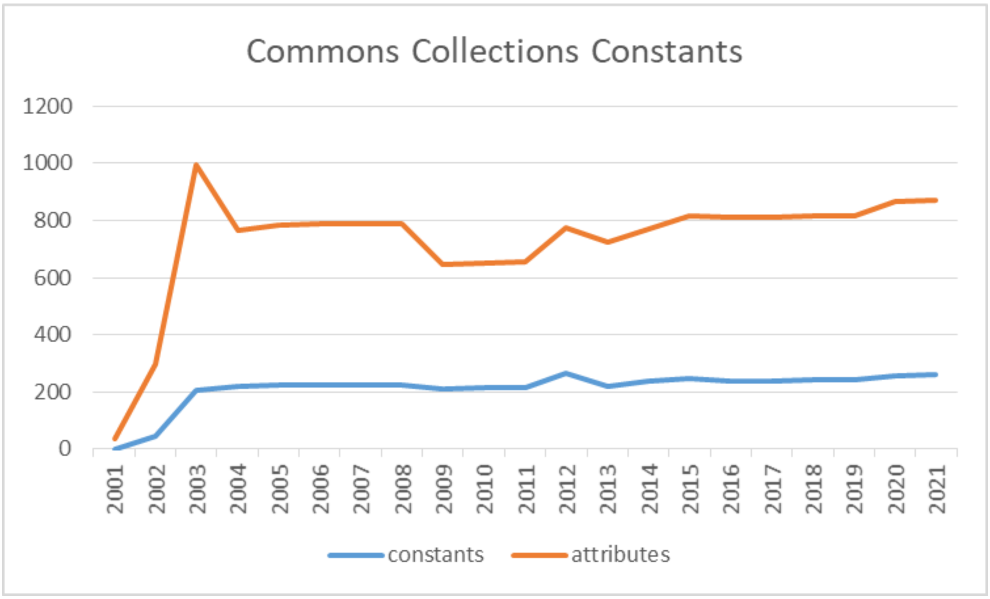

Figure 5.2.2.1: Evolution of constants for Commons Collections

Just like the total number of attributes, the number of constants increased rapidly in the first 2 years of development. At the end of 2003 there were 204 constants (20.54%) from the existing 993 attributes (the peak value in terms of number of attributes). Since then, the number of constants increased slowly until reaching the maximum value of 265 in August 2012. From there on this number fluctuated as some minor refactorings occurred within the system. It can be observed that the evolutions of the number of constants and the total number of attributes are very similar. Finally, the latest version of the system has 260 constants (29.85%) out of the 871 attributes present.

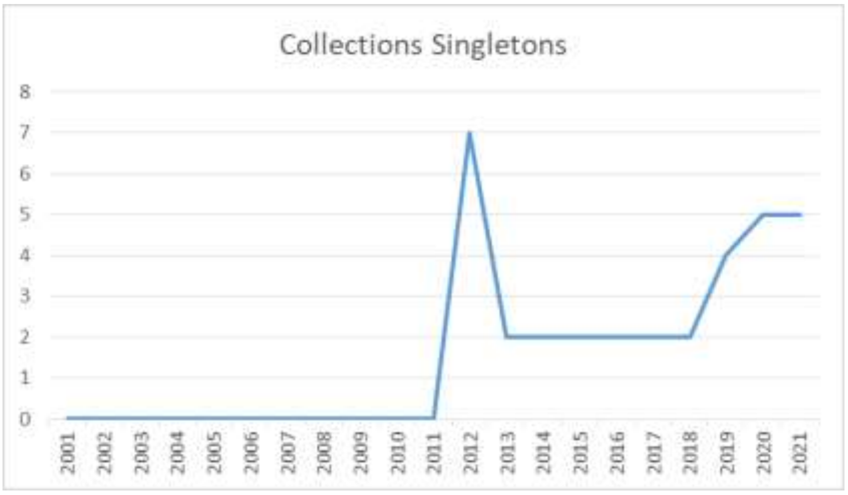

Figure 5.2.2.2: Evolution of singletons for Commons Collections

There were no singletons in the earlier versions of Commons Collections. In July 2012 7 such instances were created, all corresponding to different types of key analyzers (e.g., StringKeyAnalyzer, ByteKeyAnalyzer, or IntegerKeyAnalyzer). They were all stateless and from the Eager Instantiation category. However, most of them were part of the system for less than 1 year; in June 2013 StringKeyAnalyzer was the only singleton left. From there on the number of singletons started to increase; DefaultEquator was added in November 2013, PropertiesFactory and SortedPropertiesFactory in June 2019 and NoValuesIterator in February 2020. Similar to before, all the singletons were stateless and of type Eager Instantiation; this shows that the developers refrained themselves from creating stateful singletons.

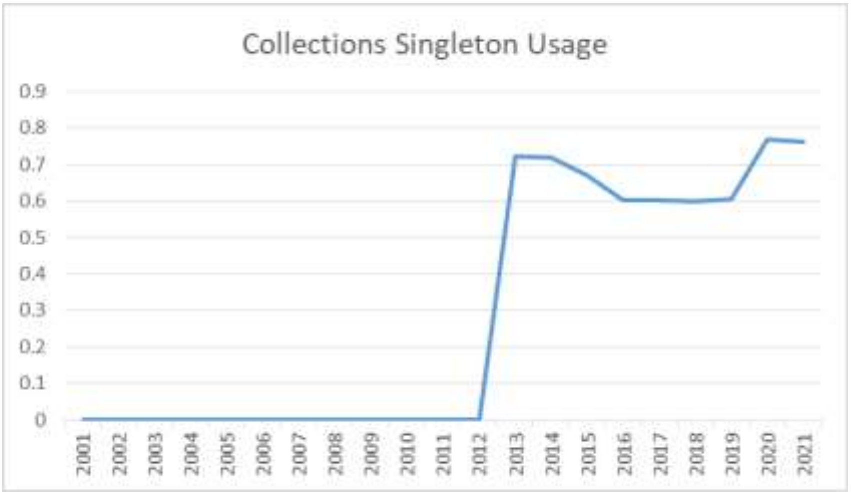

Figure 5.2.2.3: Evolution of singleton usage for Commons Collections

The usage of singletons throughout the project's lifespan is very low, less than 1% of the production classes utilize such instances. Oddly enough, no client classes were found when the first 7 singletons were added. Three clients were encountered in 2013, 1 for StringKeyAnalyzer and 2 for DefaultEquator, making the usage 0.72%. Since then the percentage started to decrease as no new clients appeared and the system was still growing in size. The only increase occurred in 2020 when the last singleton (NoValuesIterator) was created.

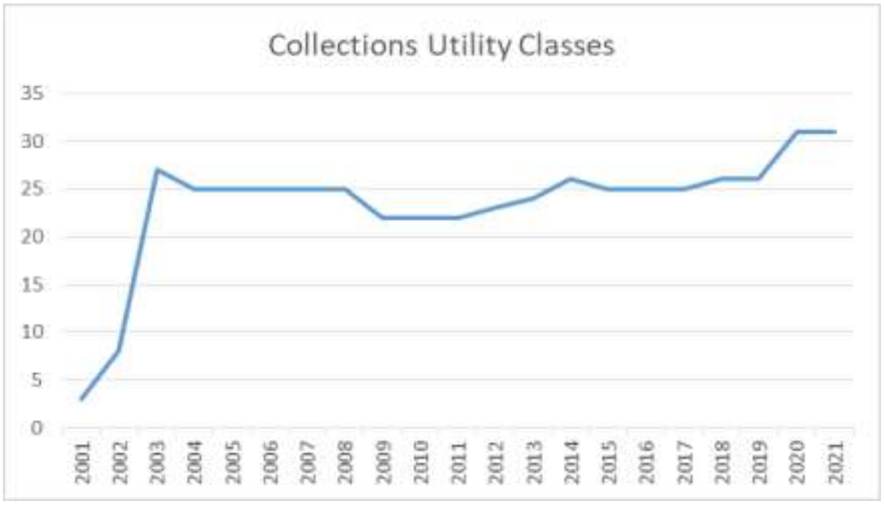

Figure 5.2.2.4: Evolution of utility classes for Commons Collections

The number of utility classes increased rapidly in the first 3 years of development; there were 3 instances in 2001, 8 in 2002, and 27 at the end of 2003. From there on this number remained rather constant between 2003 and 2019. The least amount of instances (22) were found between September 2009 and June 2012. In the last 2 years of development, the number of utility classes reached a peak value of 31.

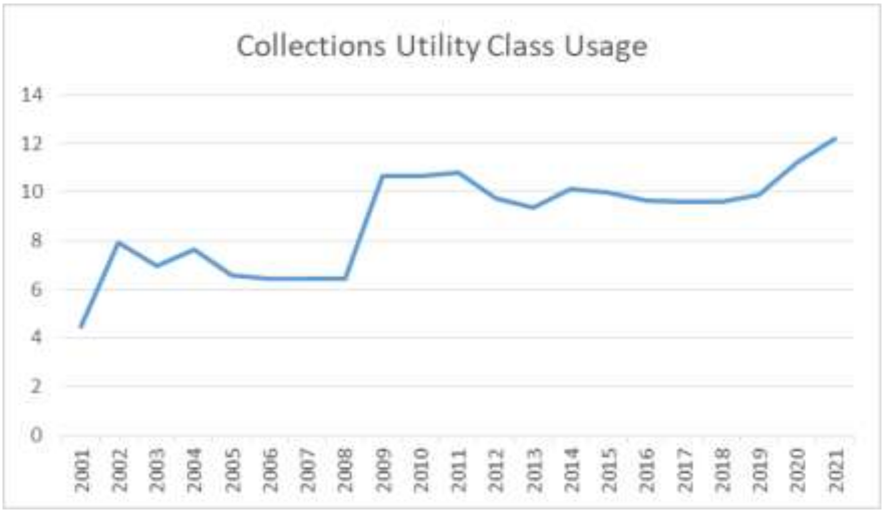

Figure 5.2.2.5: Evolution of utility class usage for Commons Collections

The usage of utility classes also increased in the earlier stages of development to around 8% between 2002 and 2004. Then we can observe a slight decrease followed by a spike to over 10% in 2009. Afterwards the usage remained constant at around 10% for almost 10 years; during this period 4 instances were added along with their client classes, but the project was also growing in size. Finally, in the last 2 years the usage increased to over 12% during a time when the number of instances reached its highest value (31).

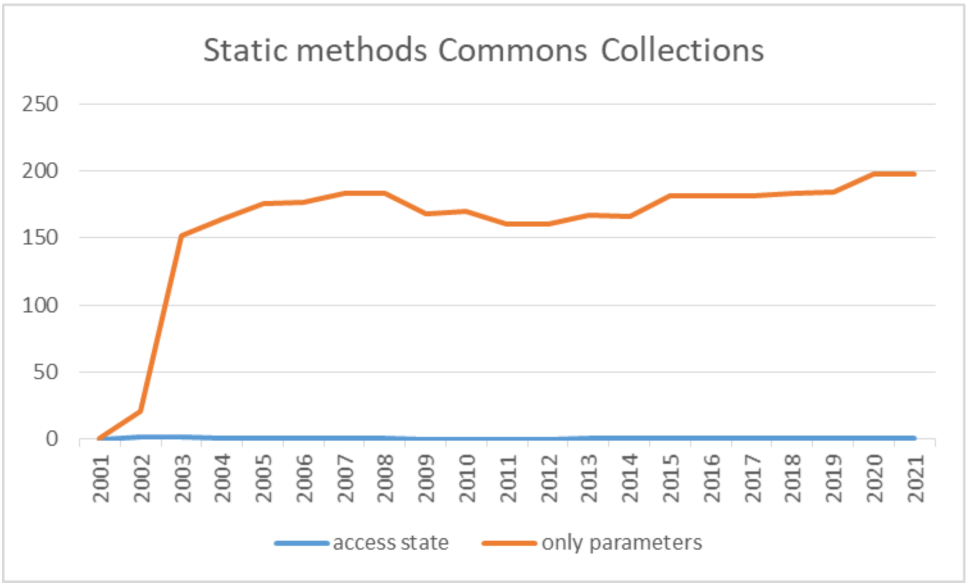

Figure 5.2.2.6: Evolution of static methods for Commons Collections

In the first version of Collection there were no static methods that access state and only 1 that solely operates on parameters. The number of instances from the first category is very low throughout the lifespan of the project. There were 2 instances between 2002-2003 and 1 between 2004-2008. No static methods that access state were found afterwards, until April 2013 when 1 such methods was added; it remained part of the system ever since. In terms of static methods that only operate on parameters, their number increased considerably in the first 3 years of development; there were 164 instances (4.78%) in June 2004. From there on, this number fluctuated until June 2015 when it reached a value of 181 (4.99%). It continued to increase in the last years of development; there are 198 static methods that solely operate on parameters in the latest version studied.

Very few static initialization blocks were found for Commons Collections throughout the project's history. One such instance was encountered in the very first commits; it was located in the BeanMap class and remained part of the system until the respective class was remove in September 2009. In May 2003 another static initialization block was used in the FunctorException class, which was deleted in May 2010. No initialization blocks were added ever since.

5.2.3. Commons Lang

The number of stat non-final attributes increased rapidly in the first year of development from 2 to 23 instances. Afterwards, this number started to decrease with some fluctuations; for example, it went from 12 in July 2003 to 18 in September 2007 and back to 12 again in August 2011. From there on the number continued to decrease until April 2017 when it reached a minimum value of 1. The respective attribute, defaultStyle, is still part of the system.

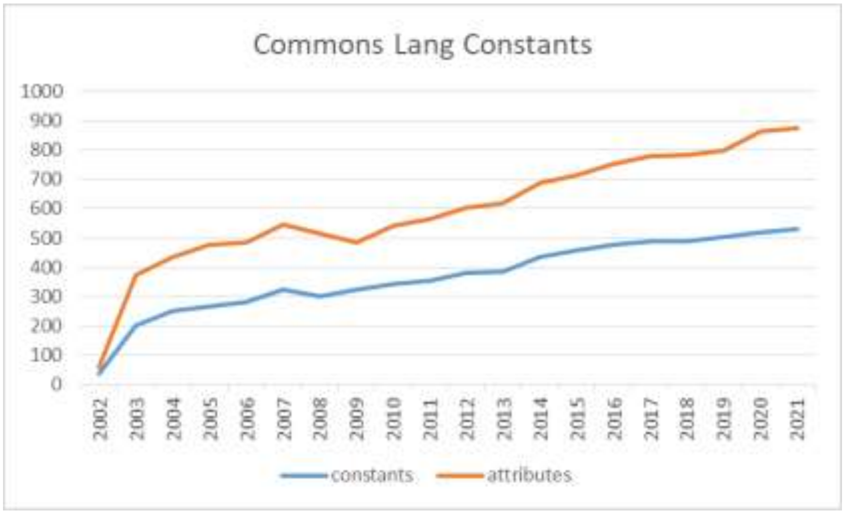

Figure 5.2.3.1: Evolution of constants for Commons Lang

The image shows that the number of constants evolves similarly to the total number of attributes. Both values grew constantly from year to year, the only exception being February 2008 when the constants dropped from 327 instances to 302 while the total number of attributes decreased from 549 to 515; during this refactoring the number of classes also went from 139 to 127. Since then the 2 values co-evolved gracefully; in the latest version studied there are 530 constants (over 60%) from a total of 875 attributes.

The only singletons found for Commons Lang, ObjectToStringComparator, was created in January 2020 and is still part of the system. It is a stateless singleton of type Eager Instantiation which does not have any clients in the latest version of the project; however, it did have 1 client when it was first added.

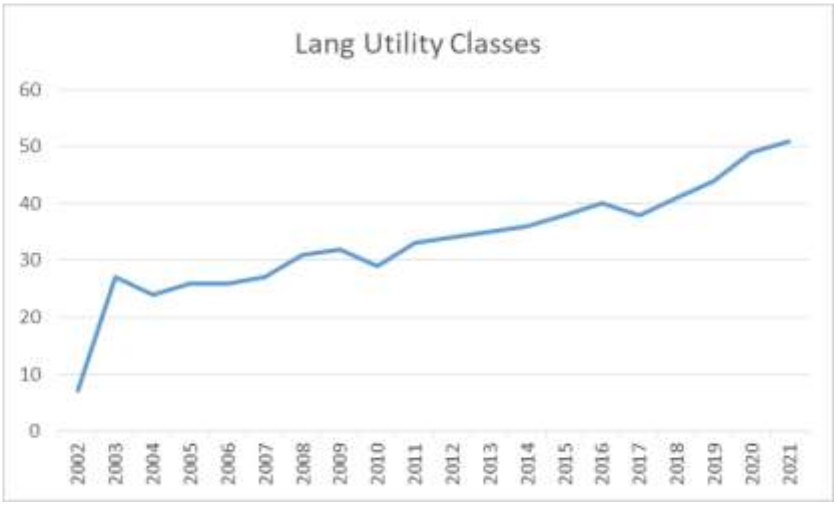

Figure 5.2.3.2: Evolution of utility classes for Commons Lang

There were 7 utility classes in the initial release of the system. Their number rose fast in the first year of development; 27 instances were found in a version from April 2003. Then it remained relatively constant the following 4 years. Afterwards, the number of utility classes increased continuously over the lifespan of the project. A maximum of 51 instances is encountered in the latest version of Lang that was studied.

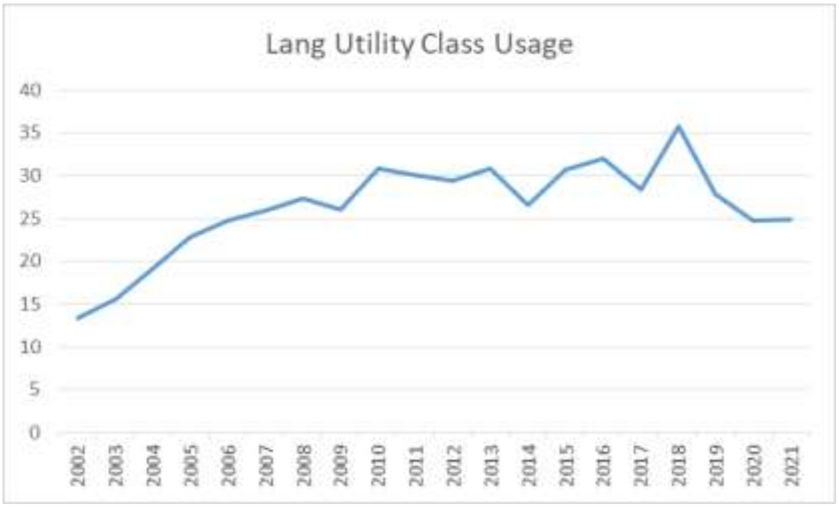

Figure 5.2.3.3: Evolution of utility class usage for Commons Lang

Utility class usage increased in the first years of development from 13.33% in the first version to 30.89% in January 2010. From there on this value fluctuates as both utility classes and other production classes are being created. In May 2018 a peak usage of 35.77% was reached; since then this value has started to decrease. The utility class usage for the latest version analysed is 24.84%.

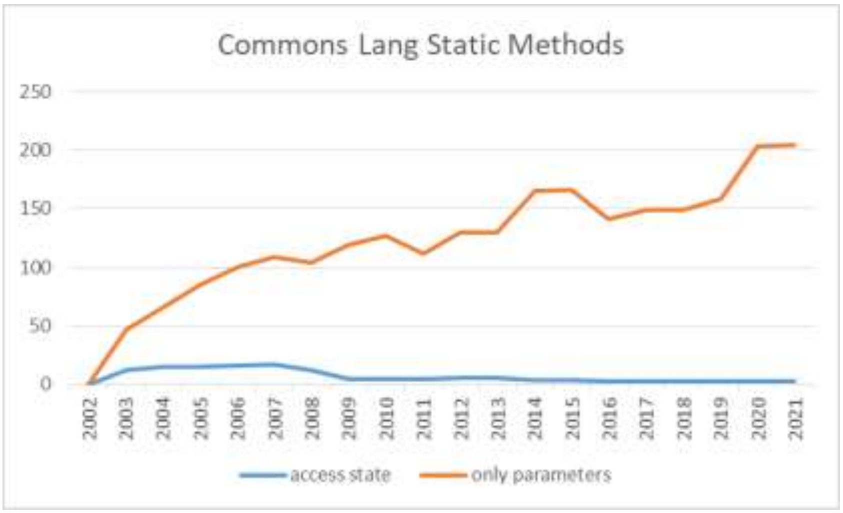

Figure 5.2.3.4: Evolution of static methods for Commons Lang

There were no static methods that were not part of utility classes in the initial version of Commons Math. The number of static methods that access state increased slightly in the first years of development until reaching a maximum of 17 (1.3%) in September 2007. From there on, it decreased to 5 (0.42%) instances in March 2009 and remained relatively constant ever since. There are 3 static methods that access state (0.08%) in the last version of the system. On the other hand, for static methods that solely operate on parameters the amount of instances grew constantly over the years. There are nonetheless 2 exceptions, 2011 when this value decreased from 127 to 112 and 2015 when it went from 166 to 141. The maximum was reached in the last year of development; 204 (5.67%) such methods were found in the latest version studied.

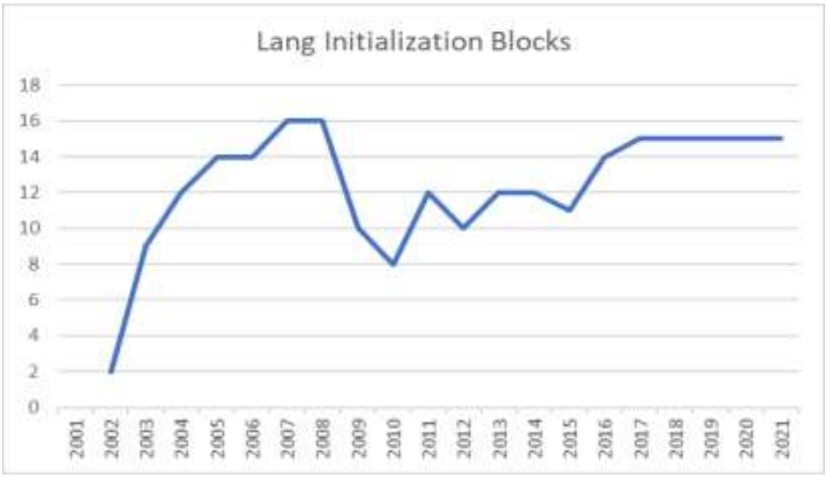

Figure 5.2.3.5: Evolution of static initialization blocks for Commons Lang

The number of initialization blocks increased rapidly in the project's first years of development. For example, there are 2 such constructs in January 2003, 5 in April, 7 in August, and 9 in December 2003. Then it continued to grow along with the system until reaching the maximum of 16 in May 2008. From there on it started to fluctuate even though the system was still growing in size. The number of instances decreased to 8 in February 2010 and then it started to increase again to 15 in March 2017; it remained constant ever since.

5.2.4. Commons Math

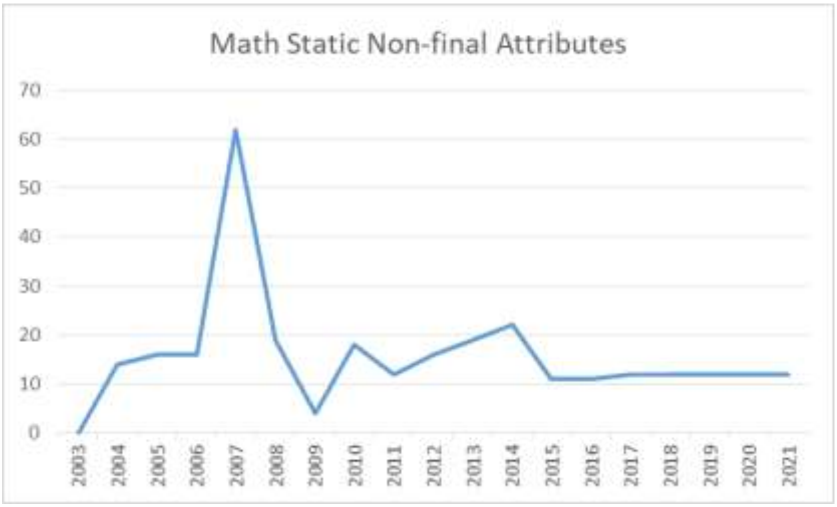

Figure 5.2.4.1: Evolution of static non-final attributes for Commons Math

There are no static non-final attributes in the initial version of Math. The number of instances increased slowly in the first 2 years of development until reaching 16 in June 2005. In February 2007 new functionalities were added to the project and the number of static non-final attributes increased almost 4 times to 61 such instances. Most of these attributes were part of the system for less than 1 year, in January 2008 only 20 were still present. From there on this number fluctuated until November 2015 when is dropped to 11. Only 1 more instance was created ever since, there are 12 in the latest version studied.

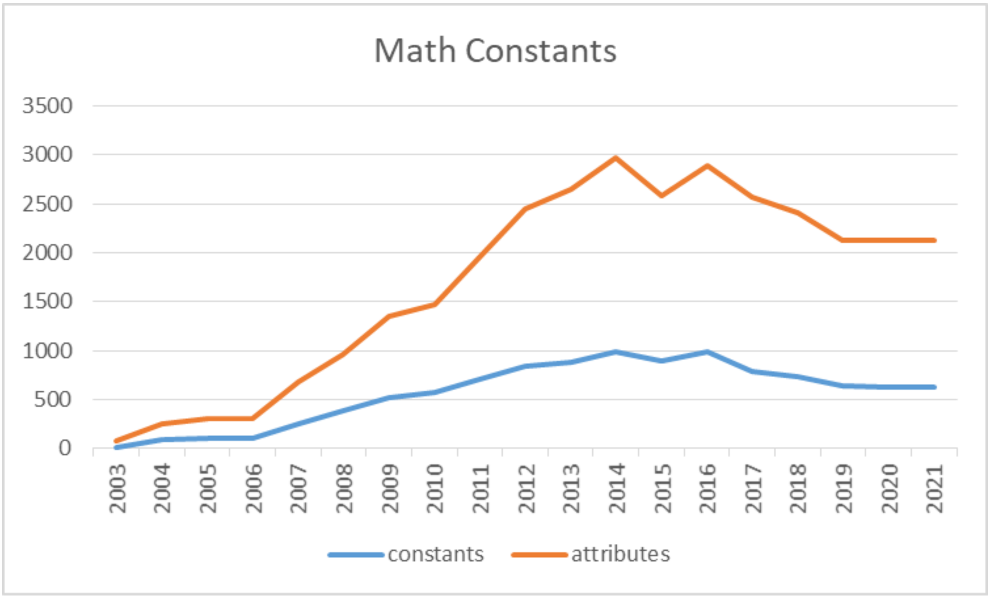

Figure 5.2.4.2: Evolution of constants for Commons Math

In the first version of Commons Math there were 13 constants (17.33%) from a total of 75 attributes. The number of instances increased over time until December 2014 when 988 constants (33.34%) were found out of 2963 attributes. Then there was a small decline followed by an increase, thereby obtaining a maximum of 993 instances in December 2016. From there on, both the number of constants and the total number of attributes decreased over time; there are 632 instances (under 30%) from 2125 attributes in the last version analysed.

The first singleton, DummyStepHandler, was created in February 2007; it was stateless, of type Lazy Instantiation, and had 2 clients (RungeKuttaIntegrator and AdaptiveStepsizeIntegrator). This instance was part of the project for 2 and a half years until it was removed in September 2009. The only other instance to ever be created, Decimal64Field, was introduced in March 2012 and is still present in the latest version studied. It is also stateless but of type Eager Instantiation and had only 1 client throughout its existence (Decimal64). Singleton usage generally peaked when an instance was created and slowly decreased as new classes were added to the project. For example, when DummyStepHandler was added it was 0.9%, then it dropped to 0 for the last commits in which the singleton was present (because it did not have any clients anymore). Similarly, the usage was 0.14% at Decimal64Field’s creation and 0.12% for the latest commit.

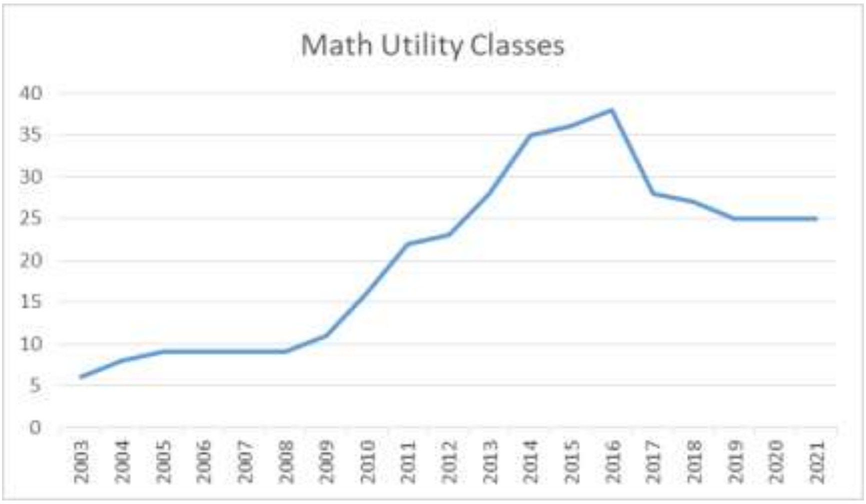

Figure 5.2.4.3: Evolution of utility classes for Commons Math

The number of utility classes was rather constant in the first 6 years of development, then it rapidly increased from 9 to 38 in the following years. The amount of instances that were added was significantly higher than the number of production classes that were created. For example, 5 new utility classes appeared between 2012 and 2013 and 7 between 2013 and 2014; during these periods around 100 production classes were developed, a number which is comparable to the ones obtained for the previous 1 year intervals (in which little to no utility classes were added). From 2016 onward the number of instances started to decrease until it reached a value of 25 for the latest version studied. This is in concordance with the refactorings that occurred in these later years of development which caused the total number of production classes to also decrease.

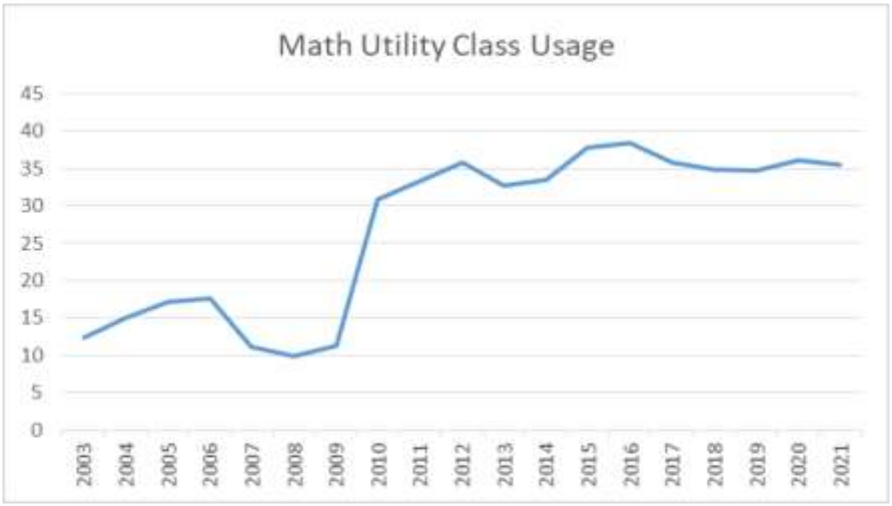

Figure 5.2.4.4: Evolution of utility class usage for Commons Math

The usage of utility classes for Commons Math is much higher compared to the other systems that were studied. Even at the beginning of the development process between 10% and 17.5% of the production classes utilized at least 1 utility class (even though there were only 6-9 instances). Since 2010 when the number of instances started to increase considerably, the percentage of client classes is much higher (over 30%). It continued to grow until reaching a maximum of 38.37% in February 2016; from there on it decreased by a small margin, but it was still around 34% to 36%. It proves once again that this type of classes are very important in projects such as Commons Math (that are structured as libraries).

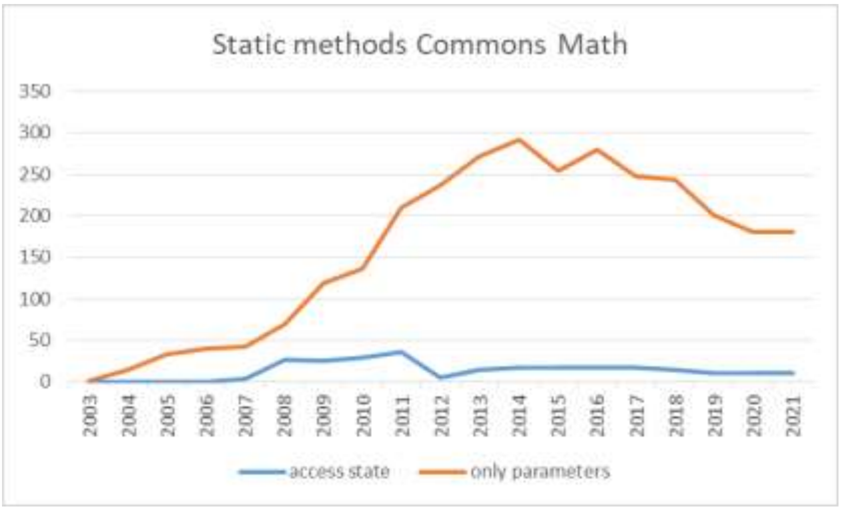

Figure 5.2.4.5: Evolution of static methods for Commons Math

The first static methods that access state (4 instances) were created in February 2007. Since then the amount of instances grew to a maximum of 36 (0.75%) in May 2011. It immediately dropped to 4 and started to increase once again. There were 14 such methods in February 2013 and 17 (0.23%) in April 2014. This number remained constant for roughly 3 years and began to decrease afterwards. There are 10 static methods that access state (0.17%) in the final version of Commons Math that was analysed. For static methods that solely operate on parameters the situation is quite different. The number of instances increased from 1 in the initial version of the project until reaching a maximum of 291 in December 2014. From there on it started to decrease, albeit with some fluctuations; there are 181 static methods that only operate on parameters (3.12%) in the latest version of the system.

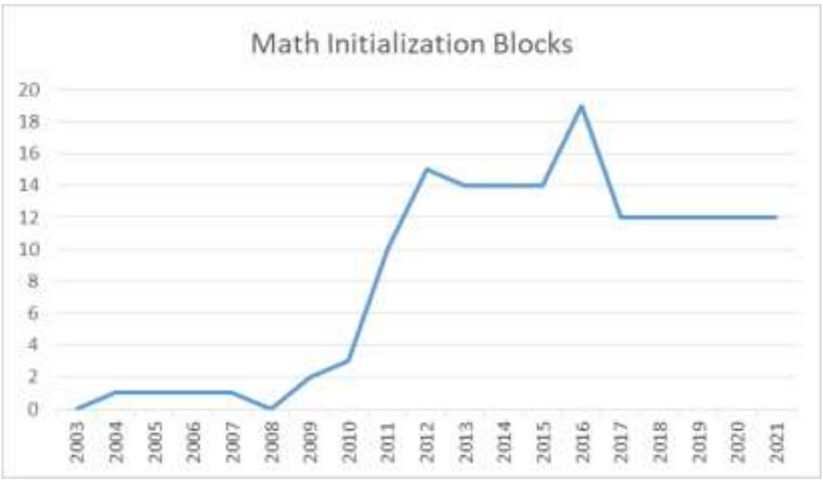

Figure 5.2.4.6: Evolution of static initialization blocks for Commons Math

Unlike what was observed thus far for this system, there were very few instances of static initialization blocks in the first 8 years of development although the project was growing rapidly. The number of instances spiked from 3 in 2010 to 10 in 2011 and to 15 in 2012. From there on it remained rather constant until 2016 when it reached the peak value of 19. Since then the system has undergone a series of refactorings, thereby reducing the amount of production classes from 1011 (January 2016) to 820 (January 2021). Unsurprisingly, the number of static initialization blocks also decreased from 19 to 12.

5.2.5. Digester

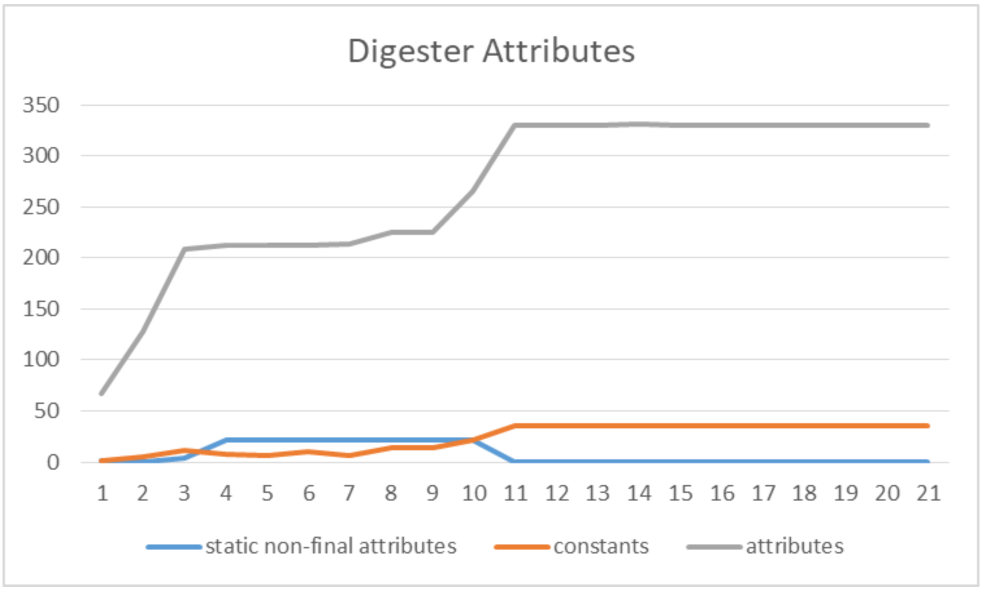

Figure 5.2.5.1: Evolution of static attributes for Digester

There was only 1 static non-final attribute in the initial version of the system, factory from the Digester class, which was removed in August 2002. Afterwards, the number of instances started to increase as more classes were being added until reaching a maximum value of 21 April 2004. This value remained constant until March 2011 when a major refactoring occurred in which all the static non-final attributes either were made final or removed. No new instances were added ever since.

The evolution of the number of constants is similar to that of the total number of attributes. There was 1 instance in the first version of Digester out of 67 attributes. Then this number started to increase with minor fluctuations until reaching 21 in August 2010. As mentioned before, a refactoring took place in the following months in which some of the static non-final attributes became constants, thus obtaining the peak value of 36 in December 2011. Since then this value remained constant as no attributes were added / removed afterwards.

Throughout the entire lifespan of the project only 1 stateless singleton was created, RuleSetCache. It appeared in August 2010 and was part of the system in that form for a couple of commits. It initially had 1 client from a total of 152 classes for the respective commit. In the commit that was studied from September 2010 it was observed that the instance was refactored into a final class; no singletons were created from there on.

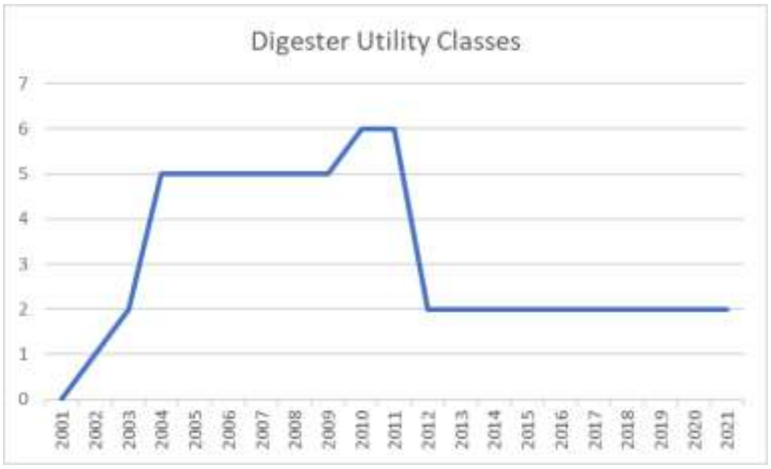

Figure 5.2.5.2: Evolution of utility classes for Digester

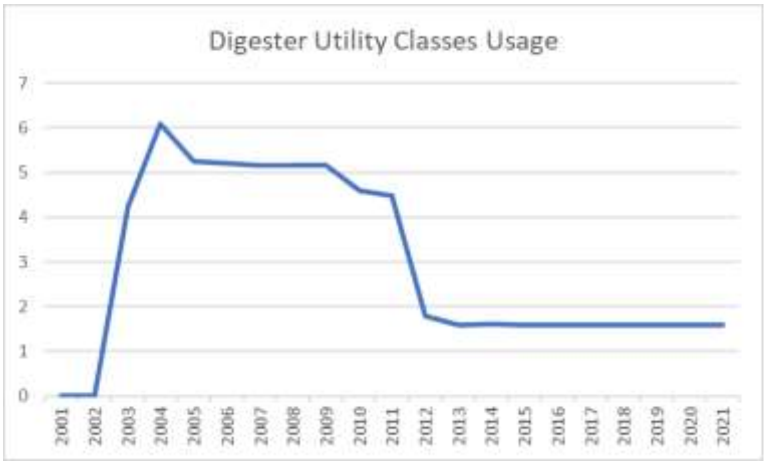

Figure 5.2.5.3: Evolution of utility class usage for Digester

The first utility class, DigesterLoader, was added to the system in December 2001. Since then, the number of instances grew constantly until reaching a maximum of 6 in August 2010. In May 2011 a series of refactorings occurred in which 4 of the utility classes were modified / removed, thereby causing the amount of instances to drop to 2; only AnnotationUtils and LogUtils remained and are still part of the project. Similar to the number of utility classes, their usage increased as more instances were added. While DigesterLoader did not have any clients initially, the percentage of production classes that utilized such instances increased over time until reaching 6.1% in January 2004. Afterwards, even though another utility class was created, the percentage decreased because significantly more classes were developed (from 97 to 152). From that point on the usage continued to drop as no new instances appeared (4 were even modified / removed) while production classes were still being created; thus, the usage finally stabilized at 1.6% from 2015 onward.

There were very few static methods which were not part of utility classes throughout the lifespan of Digester. Only 2 static methods that access state (0.35%) were encountered; they appeared between November 2003 and March 2004. For static methods that only operate on parameters, their number increased from 1 (0.57%) in the initial version of the system to 7 (0.77%) in 2011; it remained constant ever since.

Only 1 static initialization block was found in the entire history of Digester. It was added to the ParserFeatureSetterFactory class in January 2004 and remained part of the project until March 2011. As mentioned before, we did not study the clients for this type of static constructs as they are supposed to be used only for initialization purposes.

5.2.6. Geode

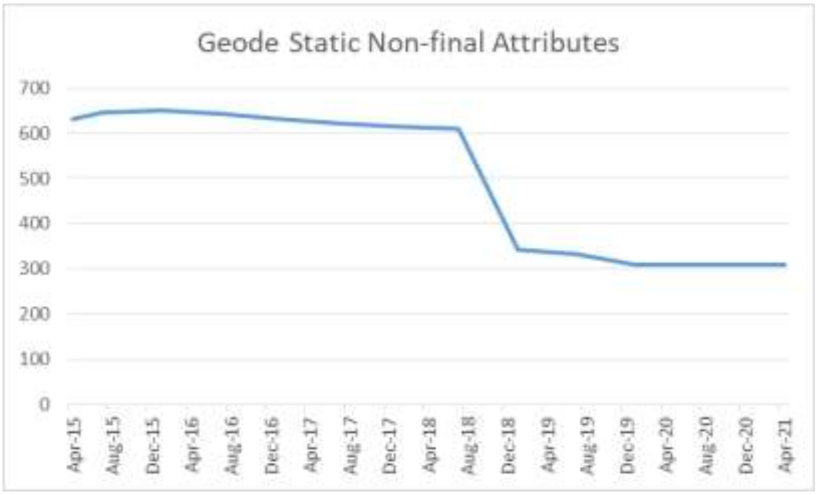

Figure 5.2.6.1: Evolution of static non-final attributes for Geode

In the first version of Geode that is available on GitHub we found 632 static non-final attributes, which is roughly 2.5% of the total number of attributes. The number of instances increased to 650 at the beginning of 2016, but then it began to decrease. Between July 2018 and January 2019 this value dropped from 611 (2.19%) to 343 (1.41%). The number of static non-final attributes continued to decrease the following year to 309 instances in January 2020. It remained almost the same ever since; there are 308 (1.39%) such attributes in the latest version of Geode.

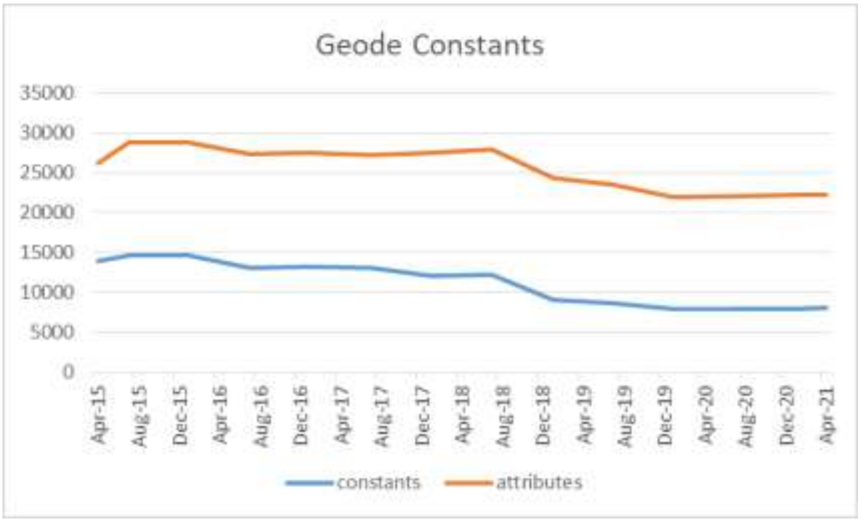

Figure 5.2.6.2: Evolution of constants for Geode

The graphs for the number of constants and the total number of attributes are very similar. Initially, there were 13954 constants (53.18%) from 26241 attributes. The number of instances increased in the first year of development to 14583 (50.68%), then it slowly decreased to 7899 (36.03%) in January 2020. From there on it grew by a very small amount; there are 8030 static non-final attributes (36.21%) out of a total of 22176.

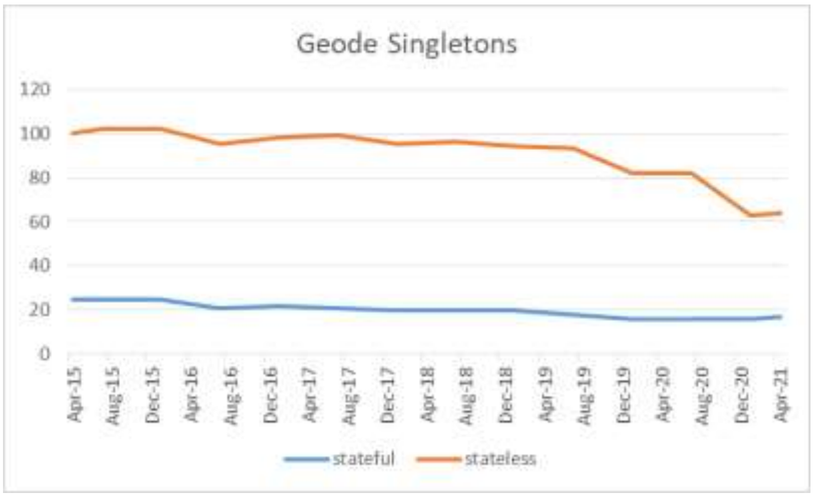

Figure 5.2.6.3: Evolution of singletons for Geode

The total number of singletons decreases as Geode evolves. There were 25 stateful instances and 100 stateless ones in the first version available on GitHub. For stateful singletons this number continuously decreased until January 2021 when there were only 16 instances. Another stateful singleton was added in February 2021, thus taking the number of instances to 17 for the latest version analysed. Similar observations can be made with regard to stateless singletons. However, there are cases in which the number increased; for example, 3 stateless singletons were created between July 2016 and January 2017 increasing the number of instances from 95 to 98. Over the years they continued to get removed; there are 64 stateless singletons in the last version of Geode.

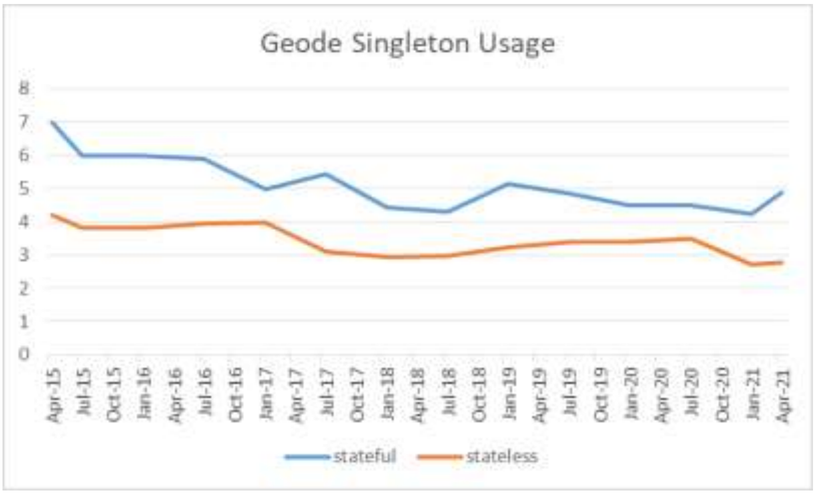

Figure 5.2.6.4: Evolution of singleton usage for Geode

In terms of singleton usage, the trend is also towards a decrease. For stateful singletons the percentage dropped from 6.98% for the first version to 4.89% for the latest version, while for the stateless ones the values are 4.18% and 2.76%, respectively. Throughout the system's lifespan there are cases in which this percentage increased (e.g., when a new instance was created or when a bunch of production classes that are not singleton clients were removed), but in general singleton usage is continuously decreasing. As an example, the percentage for the stateful variants increased from 4.31% to 5.14% between July 2018 and January 2019 even though the number of instances remained constant (20); a series of refactoring occurred during that period during which almost 200 production classes were removed.

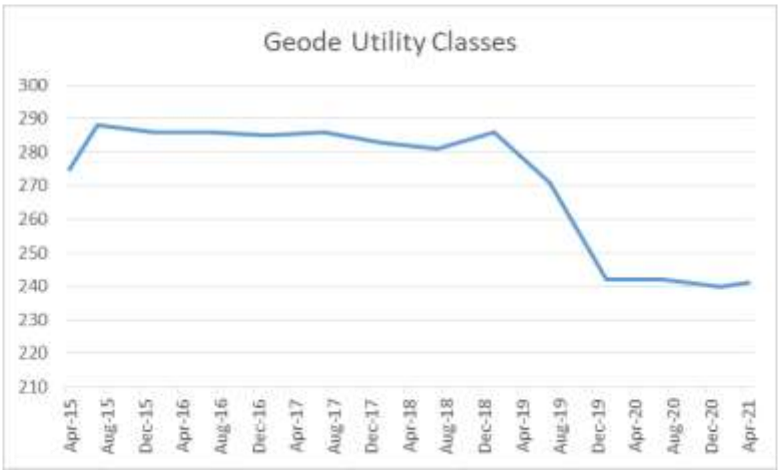

Figure 5.2.6.5: Evolution of utility classes for Geode

In the first version analysed there were 275 utility classes. This number increased in the following 6 months to 288 instances in July 2015. Then it remained relatively constant for almost 4 years; there were 286 utility classes in January 2019. From there on this value started to decrease, more abruptly at first (from 286 to 242 in less than 1 year) and slowly afterwards. In the latest version of Geode there are 241 utility classes.

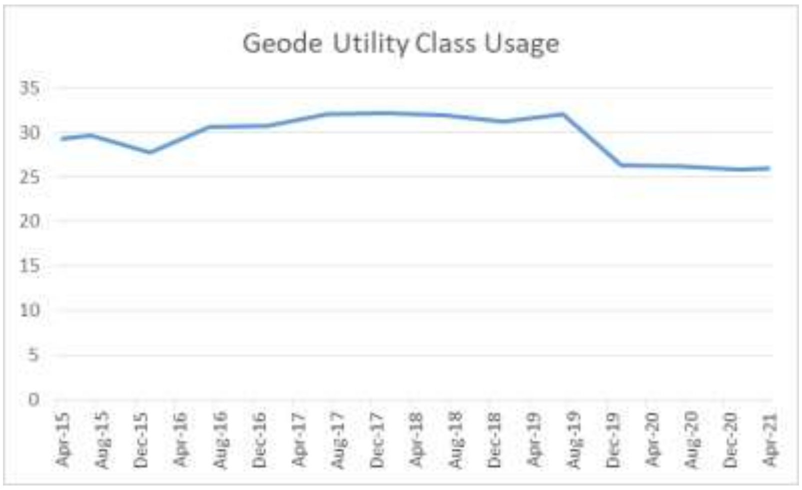

Figure 5.2.6.5: Evolution of utility class usage for Geode

The usage of utility classes fluctuates between 25% and 32% throughout the project's lifetime. It was 29.35% for the initial version studied; then it slowly increased until reaching a maximum value of 32.11% at the beginning of 2018. Utility class usage remained almost the same the following 2 years. Similar to the number of instances, the usage dropped to 26.27% at the end of 2019. It suffered only minor changes ever since; for the last version investigated the usage is 25.91%.

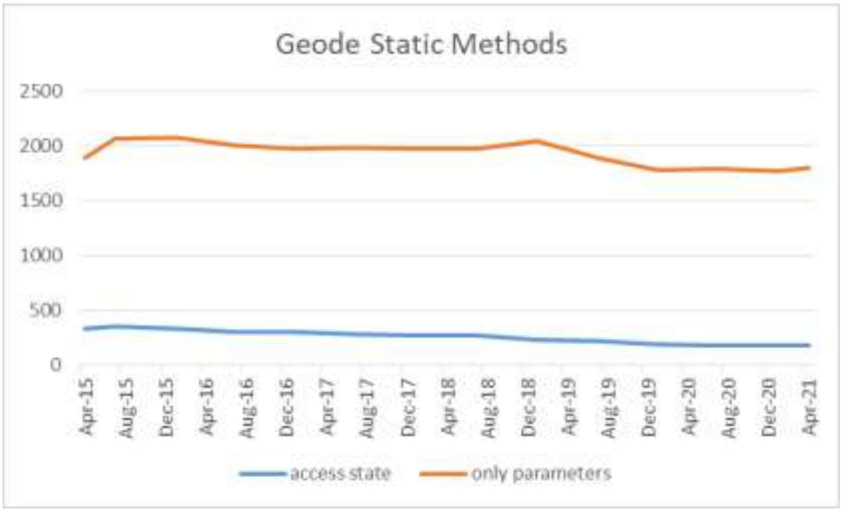

Figure 5.2.6.7: Evolution of static methods for Geode

There were 333 static methods that access state (0.67%) and 1892 that only operate on parameters (3.79%) in the first version available. For the former, the number of instances increased to 354 in August 2015, then it slowly decreased over the years; there are 184 (0.33%) such methods in the last version studied. For the second category of static methods, their amount fluctuated throughout the lifespan of Geode. The maximum of 2095 was reached in June 2015. In the latest version of the system there are 1804 (3.24%) such methods. A thing to note is that the total number of methods decreased by roughly 4000 from the initial version to the last one analysed.

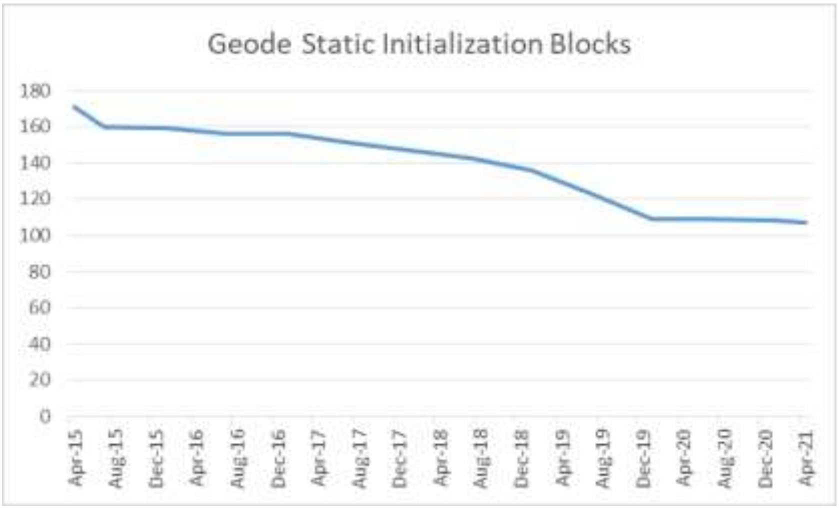

Figure 5.2.6.8: Evolution of static initialization blocks for Geode

Unlike the number of production classes which increases constantly and only drops when certain refactorings occur, for static initialization blocks we can observe a continuous decrease from the first version of the project available on GitHub to the last version studied. There were 171 instances at the beginning and 109 in January 2020. Since then this number has remained relatively constant; 107 static initialization blocks were encountered in the latest version of Geode.

5.2.7. jHotDraw

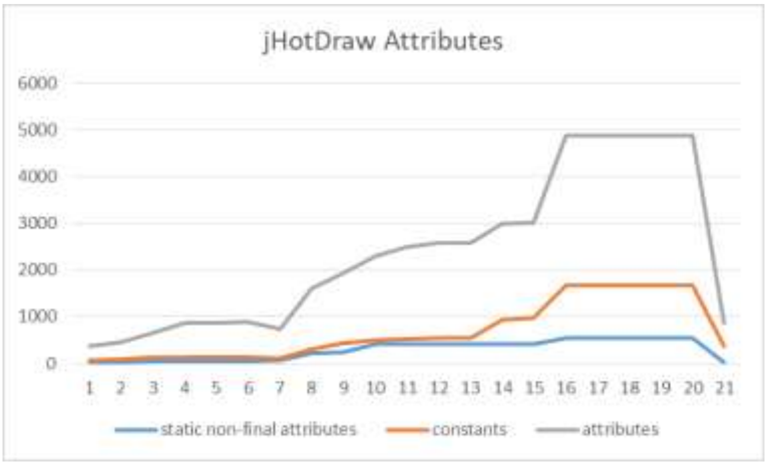

Figure 5.2.7.1: Evolution of static attributes for jHotDraw

There were 13 static non-final attributes (3.55%) and 75 constants (20.49%) in the first version of the project which had 366 attributes in total. In the first 6 years of development these numbers increased steadily; in 2006 we found 72 non-final ones (9.64%) and 109 constants (14.59%) from a total of 747 attributes, thus showing that more instances from the first category were added. The values spiked in 2007 and continued to increase rapidly until 2015 when 546 static non-final attributes (11.18%) and 1660 constants (33.99%) were present. Since then, the number of attributes suffered only minor modifications until the beginning of 2020 when they dropped to 19 for non-final attributes (2.21%) and 369 for constants (42.91%) out of a total of 860.

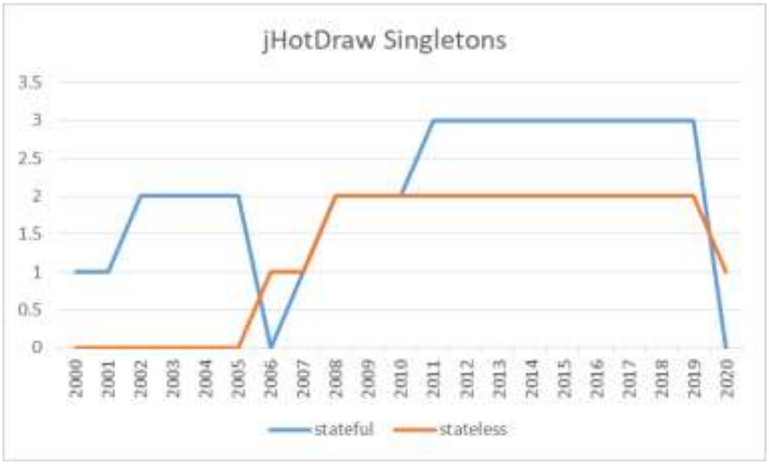

Figure 5.2.7.2: Evolution of singletons for jHotDraw

There is 1 stateful singleton in the initial version of the project, Clipboard, and no stateless ones. A second stateful singleton was added in August 2002; both instances were part of the system until 2006 when they were removed during a series of refactorings in which a stateless singleton (FigureLayerComparator) was also added. In the following 5 years 3 stateful singletons, PaletteLookAndFeel (April 2008), PaletteLabelUI (May 2008) and ActivityManager (September 2011), and 1 stateless one, PaletteButtonUI (April 2008), were added to jHotDraw. They were all part of the project until March 2020 when all but FigureLayerComparator were deleted.

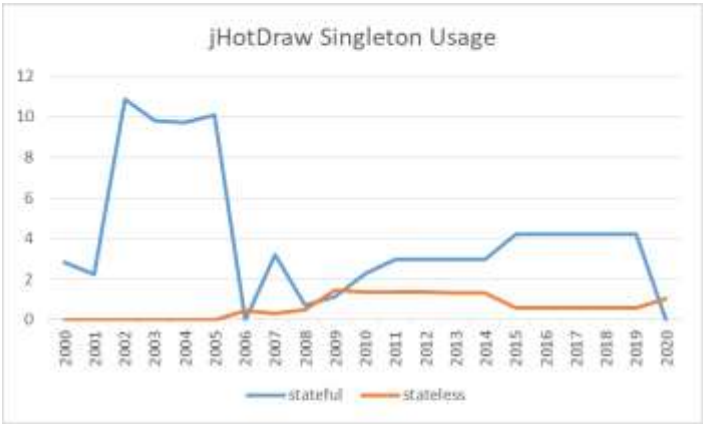

Figure 5.2.7.3: Evolution of singleton usage for jHotDraw

The usage of the 2 types of singletons varies depending on the number of instances present. For the stateful ones it spiked to 10.88% when the second instance was added; then it remained relatively constant for 4 years and dropped to 0 when both instances were removed. From there on stateful singleton usage started to increase again as 3 new instances were created subsequently. Finally, in the last year of development the usage became 0 once more because the 3 stateful singletons were deleted from the system. The usage of stateless singletons was 0 until the first instance was created in November 2006. Afterwards, it increased again once the second instance was added and slowly decreases over the years as no new instances were created while the number of production classes continued to rise. An interesting situation appeared in the final year of development when 1 of the stateless singletons was removed, but the usage increased from 0.61% to 1.03%; this is due to the fact that a considerable number of production classes were deleted during that period.

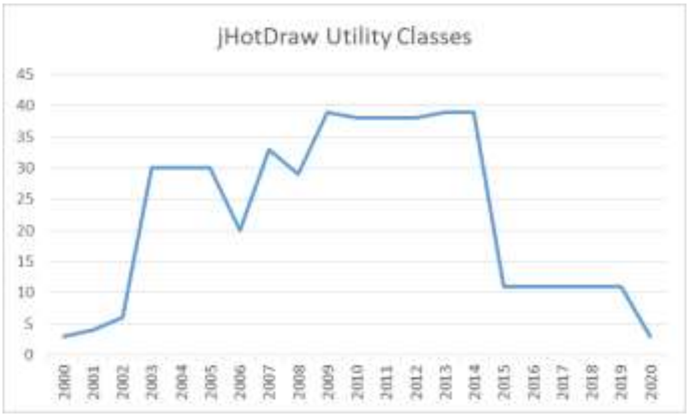

Figure 5.2.7.4: Evolution of utility classes for jHotDraw

There were 3 utility classes in the initial version of jHotDraw. The number of instances increased to 30 between 2002 and 2003, but in 2006 a series of refactorings occurred and this value dropped to 20. Afterwards it continued to increase, thereby reaching a maximum of 39 in May 2009. The number of utility classes remained relatively constant between 2009 and 2014; in 2015 a major refactoring in which almost 400 production classes were removed caused a large decrease in utility classes (from 39 instances to 11). Then no major changes were performed on the system until March 2020 when another refactoring made the number of instances drop to only 3.

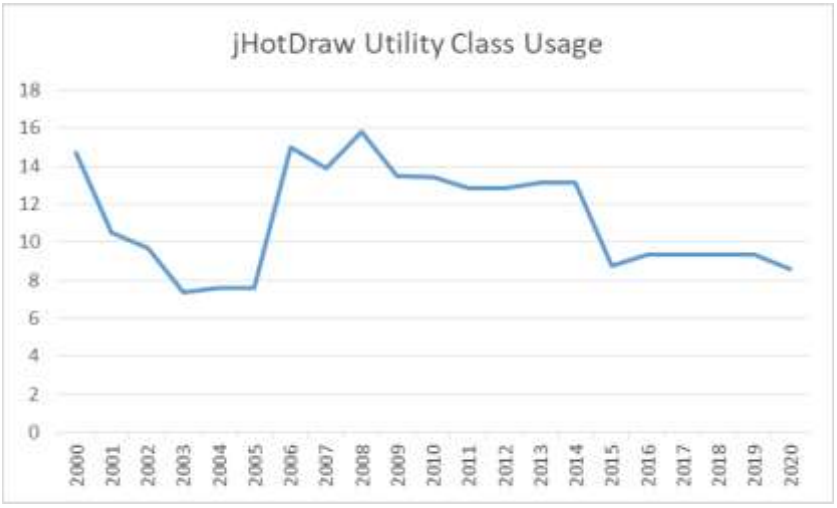

Figure 5.2.7.5: Evolution of utility class usage for jHotDraw

Although the number of utility classes increased during the first years of development, utility class usage continually decreased from 14.69% in the first version to 7.35% in September 2003. Then it started to increase until reaching 15.81% in October 2007, around the time when the maximum number of instances was obtained. From there on the usage slowly decreased over the years; in the latest version studied a usage of 8.59% was reached, which is close to the overall minimum.

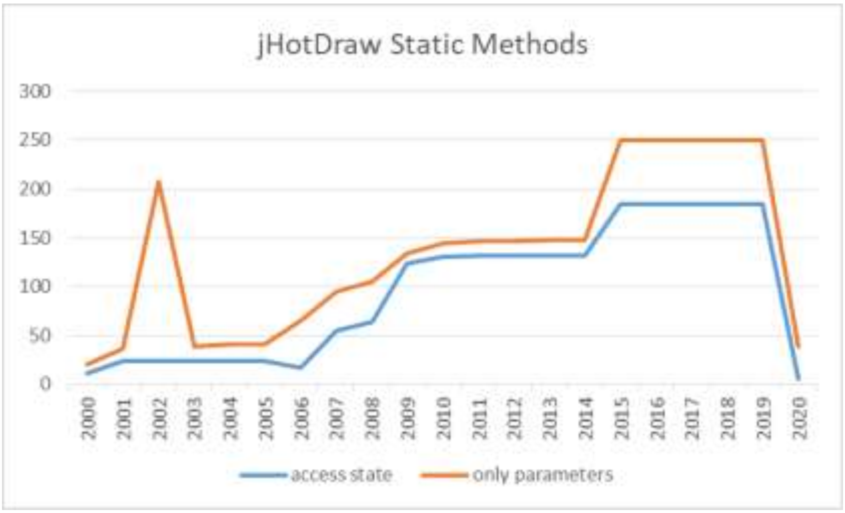

Figure 5.2.7.6: Evolution of static methods for jHotDraw

The cases for the other 2 types of static methods are very similar, but the values are a bit higher for those that solely operate on parameters. There were 11 static methods that access state (0.79%) and 20 from the latter category (1.44%) out of a total of 1392 methods. In both cases the number of instances increased over the years until reaching a maximum in the same year (2015). There were 185 (1.53%) instances from the first category and 250 (2.08%) from the second. The values remained constant for several years, then they dropped heavily; there are 6 static methods that access state (0.22%) and 39 to only operate on parameters (1.43%) in the final version of jHotDraw.

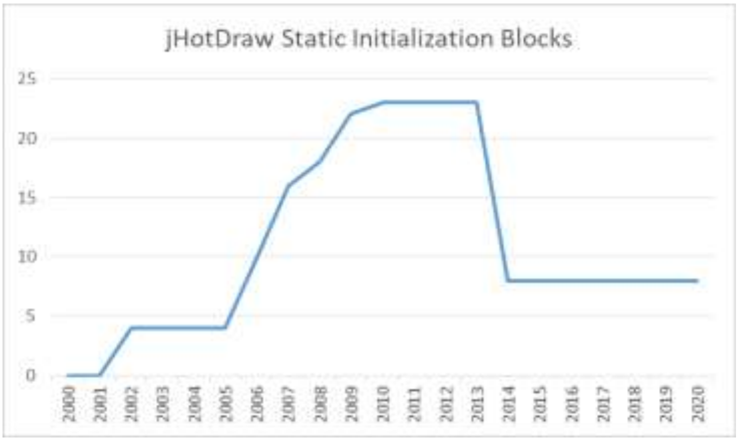

Figure 5.2.7.7: Evolution of static initialization blocks for jHotDraw

There were no static initialization blocks in the initial version of jHotDraw. The first instance was added to the TextAreaFigure class in April 2002. Since then, the number of static initialization blocks continued to increase until reaching a maximum of 23 in November 2010. It remained constant for several years until a major refactoring occurred in February 2015 and the amount of instances dropped to 8. An important observation is that the number of production classes also decreased during this refactoring from 1043 to 663. The project is rarely modified from there on, thus no static initialization blocks are added / removed ever since.

5.2.8. Pig

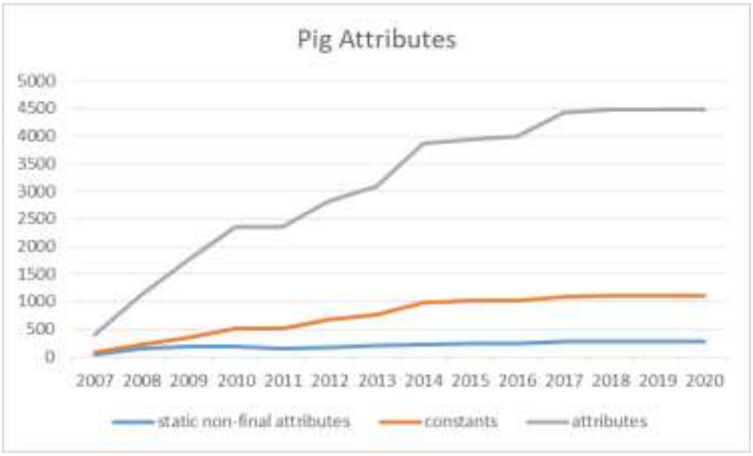

Figure 5.2.8.1: Evolution of static attributes for Pig

The number of instances for static non-final attributes, constants, and non-static attributes all increased constantly from the initial version of Pig to 2017. For example, there were 32 static non-final attributes (8.06%) and 64 constants (16.12%) in the first version studied from a total of 397 attributes. These numbers increased considerably in the first year of development; in 2008 there were 149 static non-final attributes (13.24%) and 219 constants (19.47%) out of 1125 attributes. The increase was less pronounced in the following years and these values remained almost the same in the final 3 years of development. For the last version analysed there are 269 static non-final attributes (roughly 6%) and 1101 constants (almost 25%) from the total 4480 attributes. It can be observed that the percentage of instances increases considerably throughout the project's lifespan for constants while for static non-final attributes it slightly decreases.

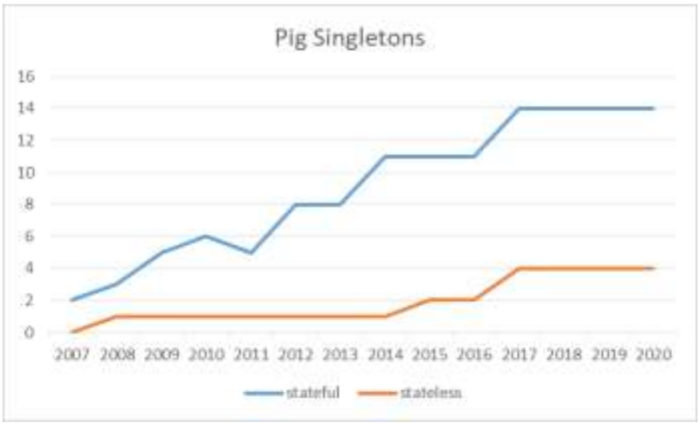

Figure 5.2.8.2: Evolution of singletons for Pig

There were 2 stateful singletons, BagFactory and PerformanceTimerFactory, and 0 stateless ones in the first version of Pig. The number of stateful instances increases over the years; there were 6 stateful singletons in 2010, 11 in 2014, and 14 in 2017. The only exception occurred in March 2011 when UDFContext was removed. In the last 3 years of development no new stateful singletons were created; 14 instances were found in the latest version studied. The first stateless singleton, TupleFactory, was added in June 2008. Since then 3 new stateless instances were created: DownloadResolver (November 2015), SparkShims (July 2017), and NonWritableTuple (August 2017); they have been part of the system ever since.

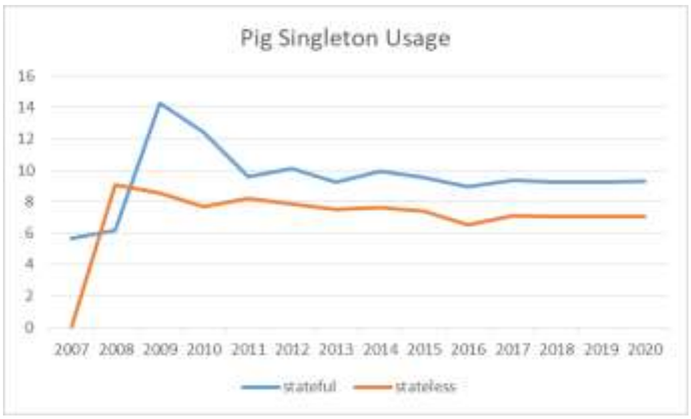

Figure 5.2.8.3: Evolution of singleton usage for Pig

In the initial version of the project stateful singleton usage was 5.65%. It quickly increased as new instances were created to a maximum of 14.75% in December 2009; then it decreased to 9.61% in 2011. From there on the value fluctuated by increasing when stateful singletons / singleton clients were added and decreasing when other types of production classes were created. For the last version investigated stateful singleton usage is 9.28%. In terms of stateless singletons, their usage is 9.09% when the first version was developed. Afterwards it started to slowly decrease as more production classes that were not stateless singleton clients were added to Pig; a minimum of 6.45% was reached in April 2016. In the last years of development stateless singleton usage increased by a small margin as 2 new instances appeared; for the latest version analysed it is 7.06%.

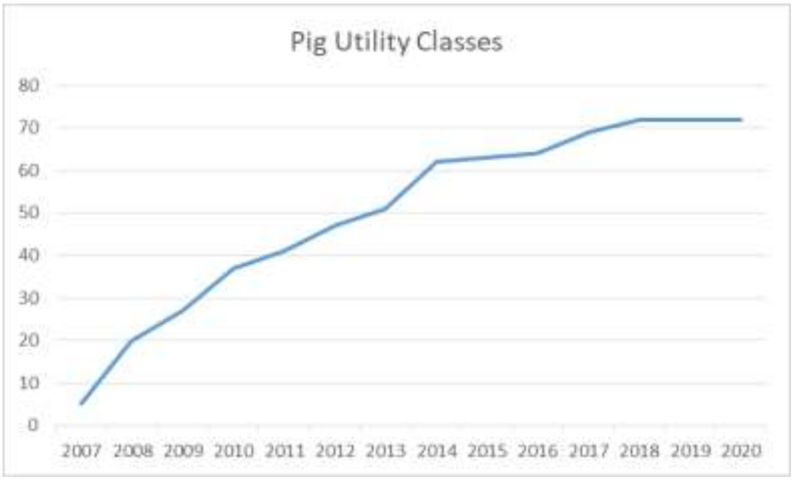

Figure 5.2.8.4: Evolution of utility classes for Pig

There were 5 utility classes in the first version of Pig. The number of instances increases throughout Pig's lifespan. At the beginning of the development cycle this increase was higher; 20 utility classes were found in 2008, 37 in 2010, and 62 in 2014. In the following years the rate of increase was considerably lower; a maximum of 72 utility classes was reached in October 2018. No instances were created ever since.

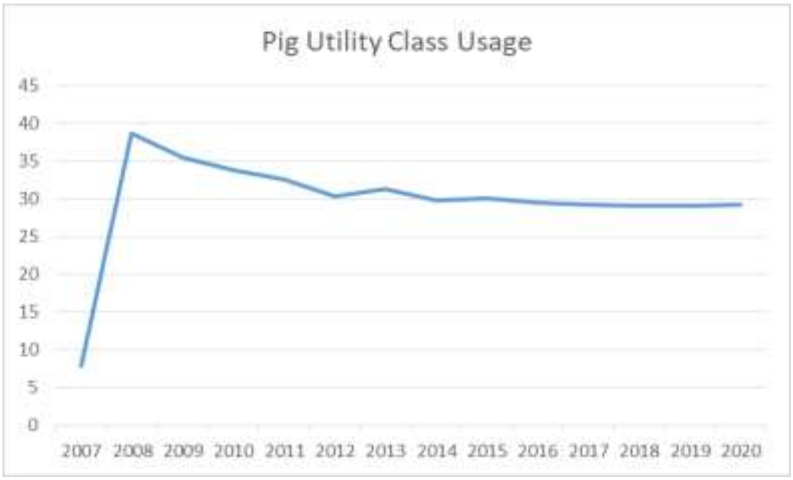

Figure 5.2.8.5: Evolution of utility class usage for Pig

In terms of usage, in increased rapidly in the first year of development from 7.91% to a peak value of 38.63%. From there one it started to slowly decrease even though new utility classes were being created. This is because the number of production classes was also rising and the classes that were added were not clients of utility classes. In the last 7 years utility class usage was around 30%; for the latest version of Pig the exact value is 29.16%.

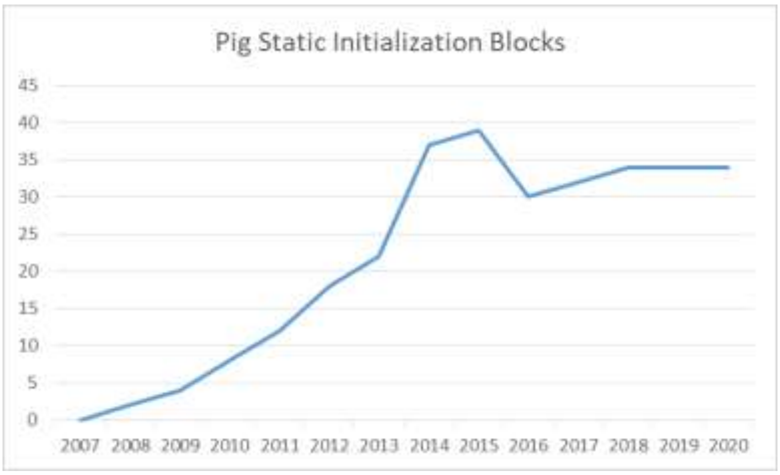

Figure 5.2.8.6: Evolution of static initialization blocks for Pig

The number of static initialization blocks increased constantly from October 2007 to September 2014 when 28 such instances were present in the production code. In October 2014 a major refactoring occurred and even though the number of classes only increased by 3, the amount of static initialization blocks went from 28 to 37 (almost 25% increase). The maximum was reached in October 2015 (39 instances), then it suddenly dropped to 30 the following month although the number of production classes continued to increase. From there it grew again until October 2018 (34 instances) and remained constant ever since.

5.2.9. Spring Core

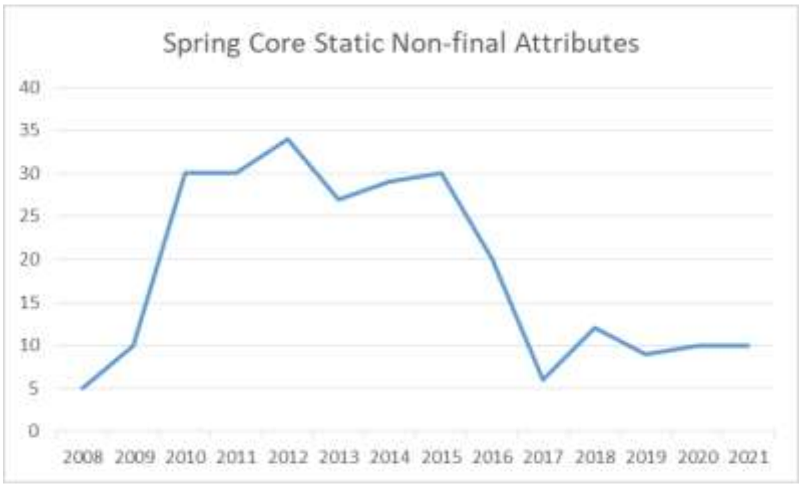

Figure 5.2.9.1: Evolution of static non-final attributes for Spring Core

There are no static non-final attributes in the first version of Spring Core. The number of instances started to increase slowly in the first 2 years of development, then it spiked to 30 in August 2010. The maximum value was reached in November 2012 when 34 such instances appeared. From there on this number first fluctuated and then it dropped to 6 in November 2017 even though the total number of attributes continued to increase. Since then the number of static non-final attributes remained fairly constant; there are 10 such instances in the latest version investigated.

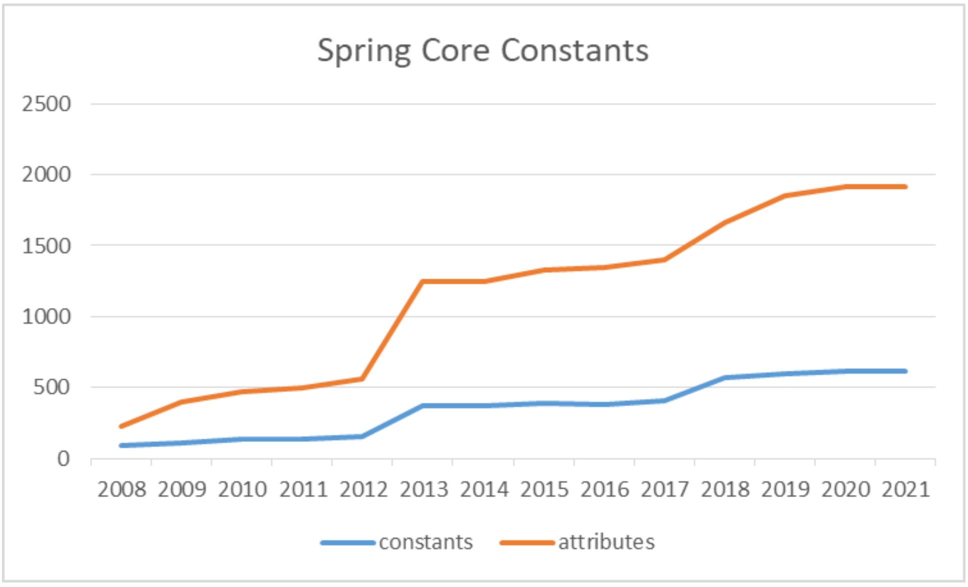

Figure 5.2.9.2: Evolution of constants for Spring Core

Unlike the static non-final attributes, both the number of constants and the total number of attributes increase constantly as the system evolves. In the first years of development constants were continuously being added. The number of instances spiked between 2012 and 2013 (from 153 to 373) as a considerable amount of functionality was added during that time period. Then it continued to increase, just like the total number of attributes. There are 611 constants (31.97%) from a total of 1911 attributes in the last version analysed.

The first 2 singletons to be added to this system, OrderComparator and StaticLabeledEnumResolver, were stateless and of type Eager Instantiation. They were introduced in February 2009 and are still part of the project. The next singletons, ComparableComparator and AnnotationAwareOrderComparator, were created in August 2012 and November 2012, respectively. They had the same characteristics as the aforementioned OrderComparator; the first had the same superclass as OrderComparator while the second directly extended it. StaticLabeledEnumResolver was removed in April 2013. Two more stateless singletons of type Eager Instantiation were added in 2014, SpringNamingPolicy and DefaultOrderProviderComparator. The latter was not part of the system for long; it was removed during a refactoring in September 2014. Finally, in May 2020 ResourcePropertiesPersister was added to Spring Core; it is the fifth stateless singleton of type Eager Instantiation that was identified in the latest version of the project (which was studied in the previous Section).

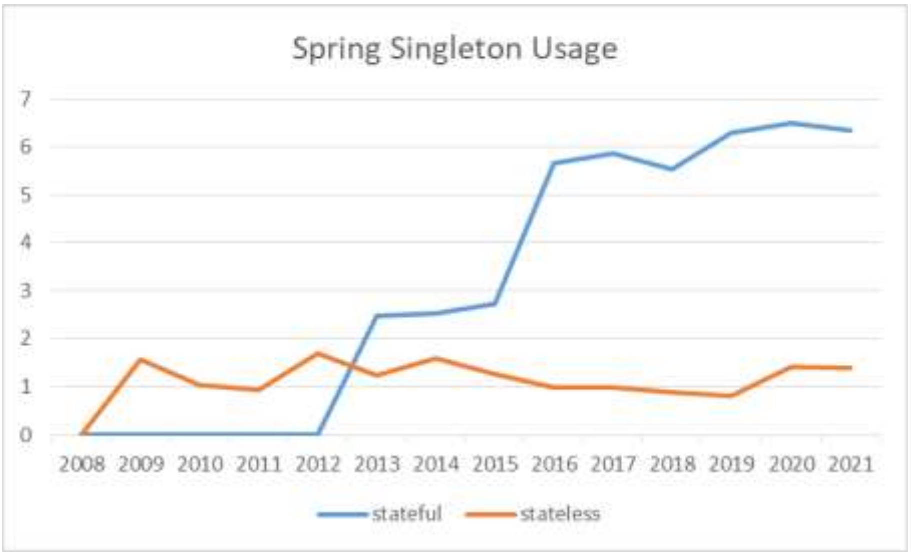

Figure 5.2.9.3: Evolution of singleton usage for Spring Core

For stateful singletons the usage was 2.49% in July 2013 when the first instance was created. Then it continued to rise as more clients of the respective class were added to the project. The highest usage (6.51%) occurred 1 month after the second instance appeared; for the final version investigated stateful singleton usage is 6.35%. In terms of stateless singletons, it started at 1.57% when the first 2 instances were created. Then it fluctuates between 1% and 2%; even though 3 more instances were added, very few of the production classes that were created subsequently are clients of the existing stateless singletons. For the latest version of Spring Core the usage is 1.39%.

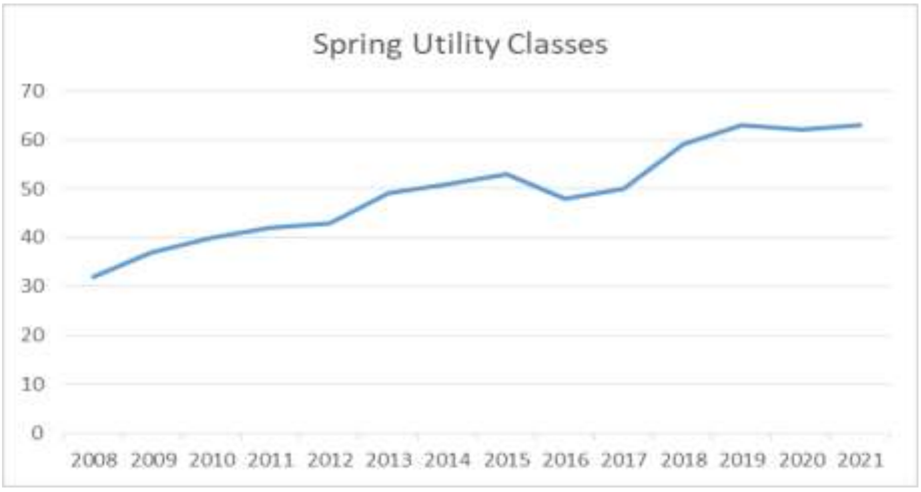

Figure 5.2.9.4: Evolution of utility classes for Spring Core

There were 32 utility classes in the initial version that was studied. The number of instances increases almost linearly until reaching 63 for the latest version of Spring. During the entire period there were only minor fluctuations, the most notable one being between 2015 and 2016. On a closer inspection it was observed that a major refactoring occurred during that time which caused the total number of production classes to drop from 4745 to 4085. Utility classes have continued to be added after that event, thereby obtaining a maximum of 63 instances in March 2019.

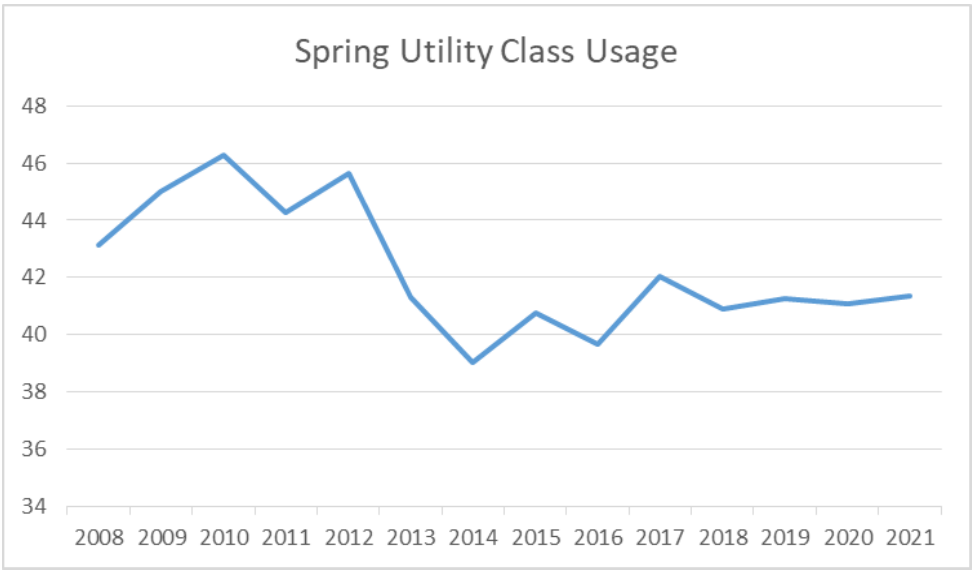

Figure 5.2.9.5: Evolution of utility class usage for Spring Core

The usage of utility classes is high throughout the project's history (roughly between 39% and 46%). However, we can observe that it was a bit higher towards the beginning of the development process. It started at over 43% and reached the peak value of 46.27% in September 2010. Then it decreased by a few percentages to a minimum of 39.03% in Match 2014. From there on it fluctuated for several years until finally stabilizing at around 41% in 2019.

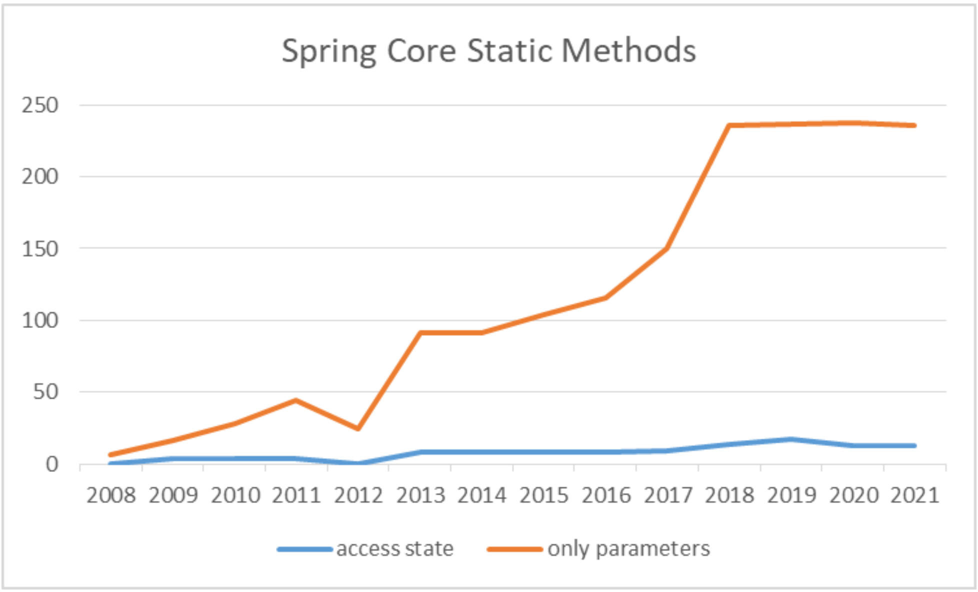

Figure 5.2.9.6: Evolution of static methods for Spring Core

There were no static methods that access state and 6 which solely operate on parameters (0.43%) in the first version of Spring Core that contained code. Only 4 instances from the first category were created until 2013. From there on, their amount started to slowly increase until reaching a maximum of 17 in March 2019. In the last version studied there are 13 (0.27%) such methods. For static methods that only operate on parameters, the number of instances increased more heavily (except in 2012 when a major refactoring occurred) until reaching the value of 236 in September 2018. It remained almost the same ever since; there are also 236 static methods of this kind (4.89%) in the latest version analysed.

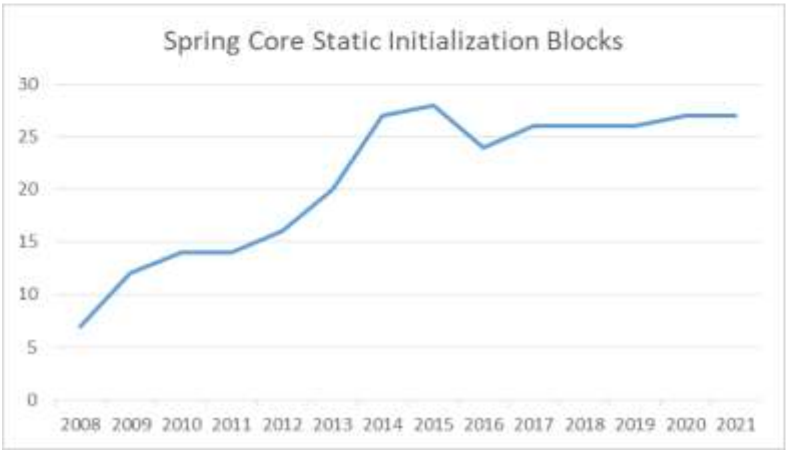

Figure 5.2.9.7: Evolution of static initialization blocks for Spring Core

The number of static initialization blocks grew quite constantly in the first half of the development period from 7 instances in October 2008 to 28 in December 2015. From there on the amount of initialization blocks remains more or less the same until the final version investigated (April 2021, 27 instances). However, during these last years of development, only a relatively small number of production classes were added; there were 4745 in January 2016 and now there are 4827 classes.

5.2.10. Tomcat

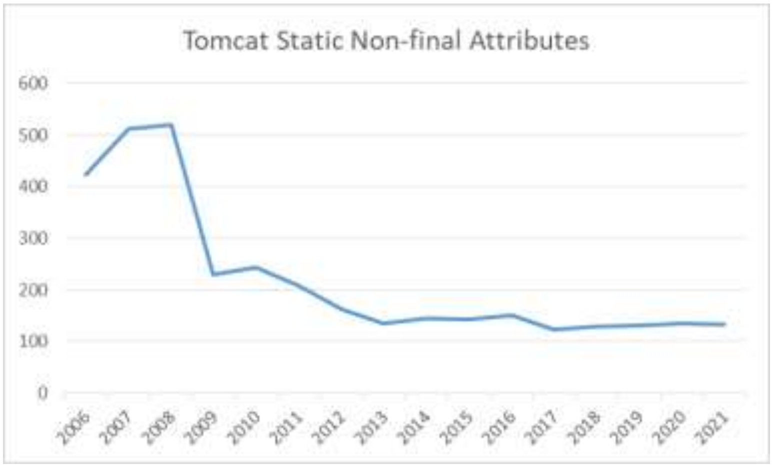

Figure 5.2.10.1: Evolution of static non-final attributes for Tomcat

There were 424 instances of static non-final attributes in the first version analysed. First, this number increased to a maximum of 520 in October 2008, then it dropped to 229 in November 2009 even though the number of constants and the total number of attributes were growing. The number of instances continued to decrease over the years and remained fairly constant in the last 4 years of development; there are 134 static non-final attributes in the latest version of Tomcat.

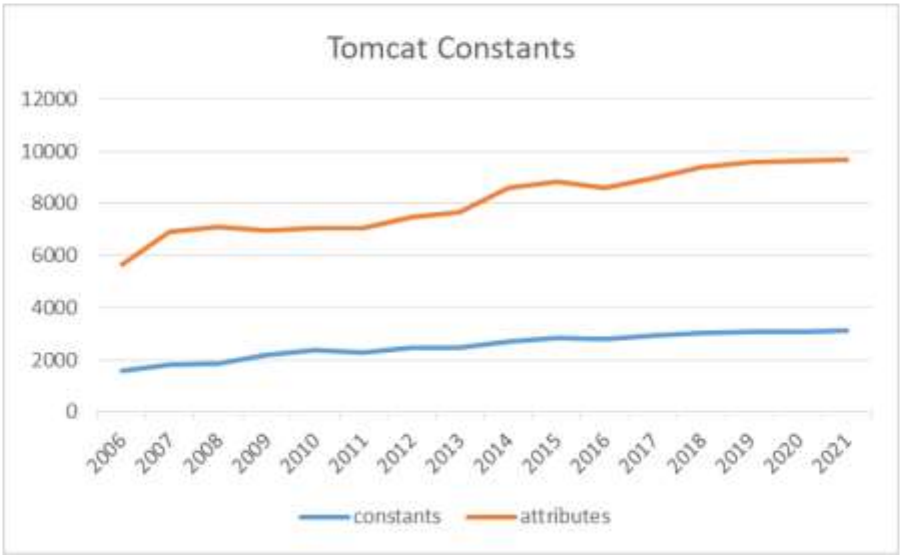

Figure 5.2.10.2: Evolution of constants for Tomcat

Just as the total number of attributes, the number of constants increases slowly as Tomcat evolves. There were 1591 instances (28.14%) from a total of 5653 attributes in the first version of the project. In the last version that was analysed there are 3106 constants (32.18%) out of 9652 attributes. The percentages are very similar throughout the entire lifespan of the system.

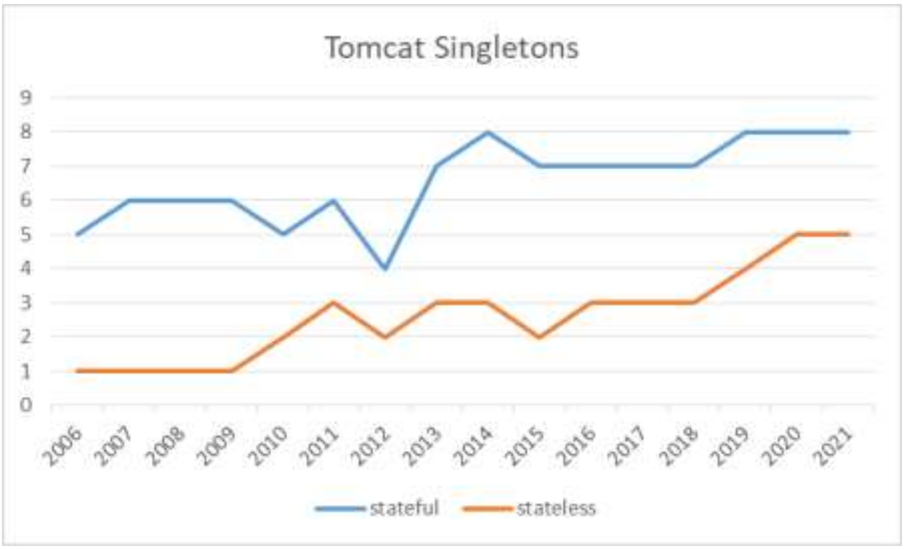

Figure 5.2.10.3: Evolution of singletons for Tomcat

Throughout Tomcat's lifespan there are very few singletons considering the size of the system. In the initial version of the project there were 5 stateful singletons and a stateless one. The number of stateful instances fluctuated in the first 6 years of development; 4 were found in October 2012 and then the amount increased to 8 in May 2014. From there on 1 instance was removed in September 2015 while another was added in April 2019; the number of stateful singletons has not changed ever since.

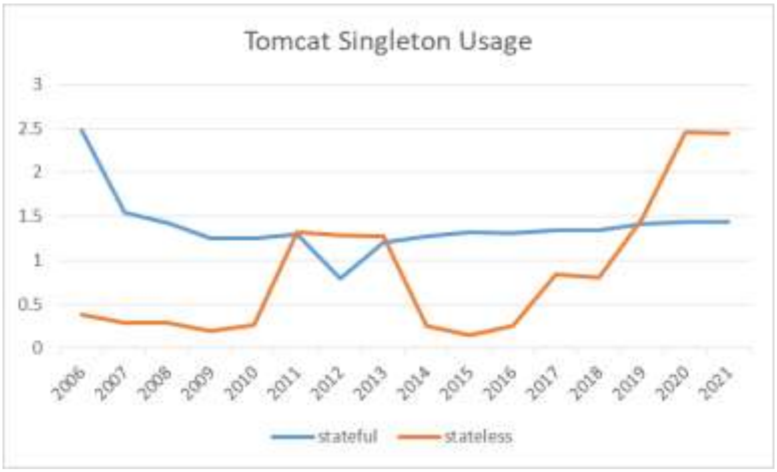

Figure 5.2.10.4: Evolution of singleton usage for Tomcat

The usage of both types of singletons is also low for this project (less than 2.5%). For the stateful ones it was 2.47% initially and then it continuously decreased until reaching a minimum of 0.76% in January 2013 (when there were only 4 such instances). Since then it began to slowly increase over the years as new stateful singletons / clients were created; for the latest version of Tomcat stateful singleton usage is 1.44%. For the stateless variants it is the other way around; usage was very low at first and grew over the years, although not by a significant amount. In the first version of the system this value was 0.38%. It remained relatively constant in the first 4 years of development and then spiked to 1.32% in December 2011. The only decrease occurred between 2013 and 2015 when a minimum of 0.15% was reached. Stateless singleton usage increased in the final years of development. A peak value of 2.45% was obtained in December 2020; not much has changed since.

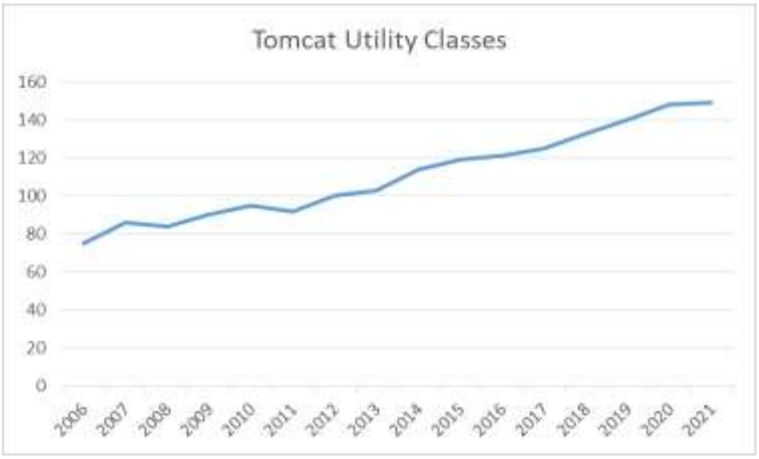

Figure 5.2.10.5: Evolution of utility classes for Tomcat

There were 75 utility classes in the first version of Tomcat. The number of instances constantly increases throughout the system's lifespan. The only exceptions occurred between 2007-2008 and 2010-2011 when this number decreased from 86 to 84 and from 95 to 92, respectively. Since then the value only increased; there are 149 utility classes in the latest version investigated.

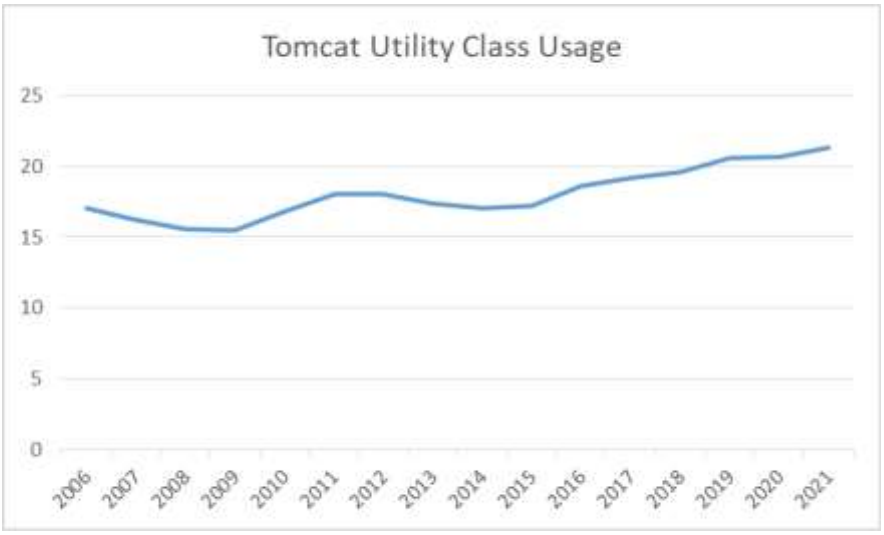

Figure 5.2.10.6: Evolution of utility class usage for Tomcat

For the initial version of the project utility class usage was 17.01%. In the following 3 years this value decreased to 15.46% as instances were added / removed while the number of production classes was always growing. A similar situation can be observed between 2011 and 2014; although instances were created, many more production classes that were not clients of utility classes were added to the system. From there on the usage grew constantly in connection with the number of utility classes that were created; it is 21.35% in the last version of Tomcat that was analysed.

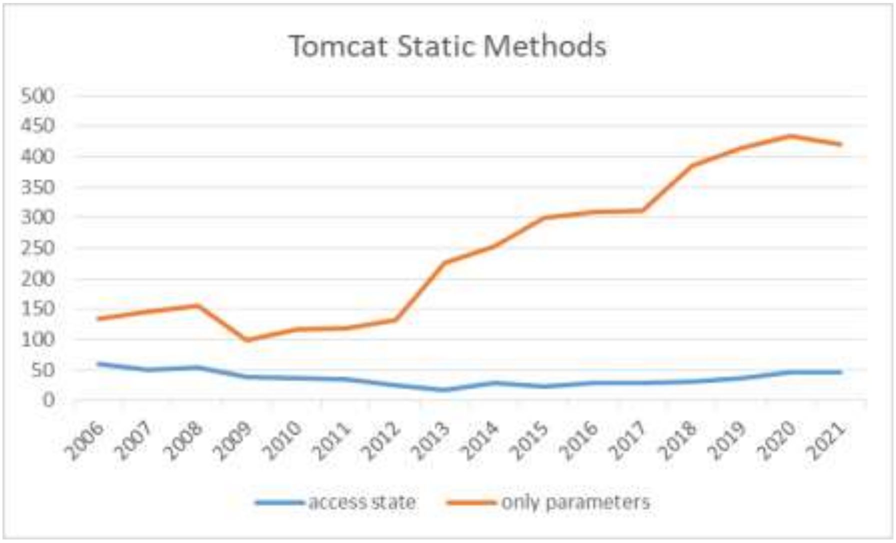

Figure 5.2.10.7: Evolution of static methods for Tomcat

There were 60 static methods that access state (1.85%) and 134 which only operate on parameters (4.13%) out of a total of 3246 methods. The number of instances for the former decreased until halfway through the development period; 18 such instances appeared in a version from March 2013. From there on, the value slowly increased; there are 46 methods that access state (0.22%) in the latest version of Tomcat. For the static methods from the second category, the corresponding value saw a small increase followed by a decrease until reaching a minimum of 100 (2.75%) instances in November 2009. Since then the number of static methods that solely operate on parameters increased continuously; a maximum of 434 was found in December 2020. In the last version of the project there are 420 (1.98%) such instances.

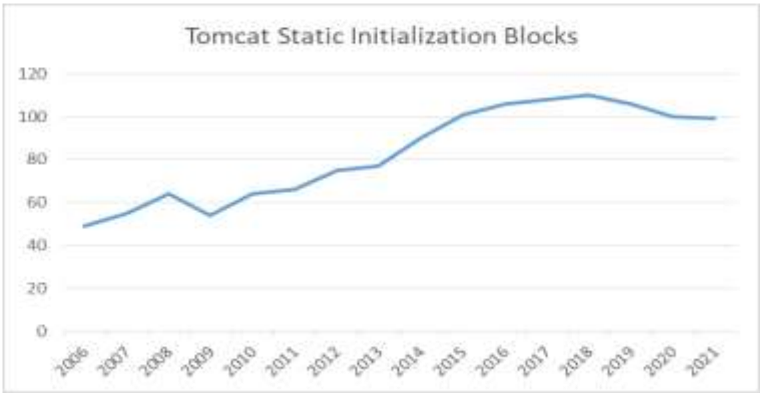

Figure 5.2.10.8: Evolution of static initialization blocks for Tomcat

With the exception of 2009, static initialization blocks were constantly being added to Tomcat until October 2018. The number of instances was 49 in the first version that was studied and peaked at 110. During this period roughly 100 production classes were created per year, some of which contained initialization blocks. From 2018 onward the number of instances started to slowly decrease; there are 99 in the latest version which was investigated. However, the number of production classes is also smaller (2126 vs. 2160 in 2019).

5.2.11. Wicket

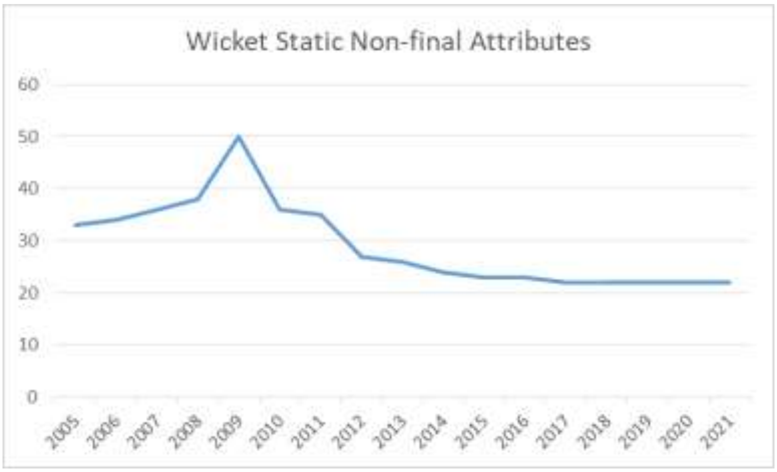

Figure 5.2.11.1: Evolution of static non-final attributes for Wicket

The number of static non-final attributes increased in the first 4 years of development from 33 until reaching a maximum value of 50 in October 2009. Afterwards, it dropped to 27 in November 2012 and remained fairly constant ever since; from July 2017 onward there are 22 static non-final attributes in Wicket's classes.

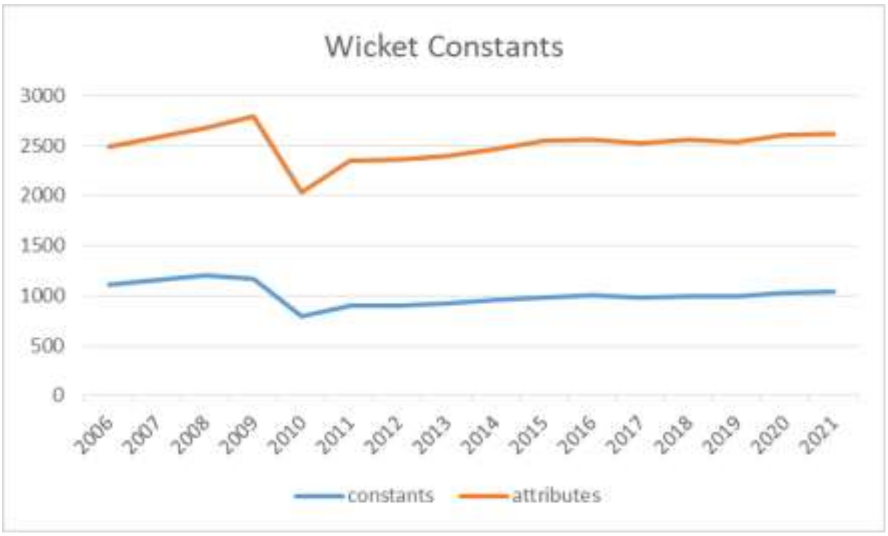

Figure 5.2.11.2: Evolution of constants for Wicket

The evolution of the number of constants looks very similar to that of the total number of attributes. The amount of instances increased in the first years of development until reaching a maximum of 1204 in December 2008. Then a series of refactorings occurred which caused both the number of constants and the total number of attributes to decrease significantly (from 1163 to 793 and from 2789 to 2032, respectively). Since then both values have increased steadily over the years; there are 1036 constants (almost 40%) from a total of 2617 attributes.

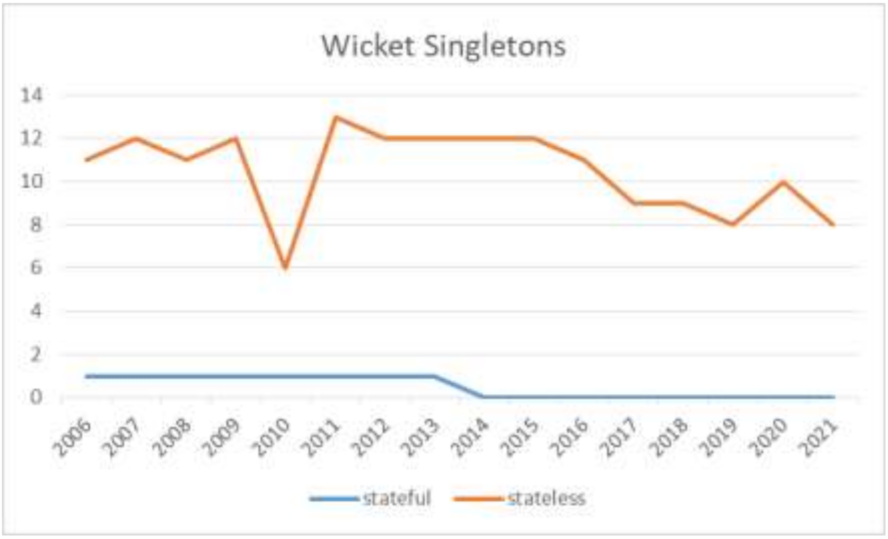

Figure 5.2.11.3: Evolution of singletons for Wicket

There is 1 stateful singleton, Result, and 11 stateless ones in the first version of Wicket in which the code was added. The stateful singleton was removed in July 2014 and no instances of this type have been created ever since. In the following 3 years 2 new instances were added and 1 was removed. In March 2010 the number of stateless singletons drops to 6 and then it spikes to a maximum of 13 in November 2011. From there on the number of instances remained relatively constant the following 5 years. Afterwards it started to decrease with one exception, the addition of PageViewCSSResourceReference and WicketCoreCSSResourceReference in January 2020. There are 8 stateless singletons in the latest version studied.

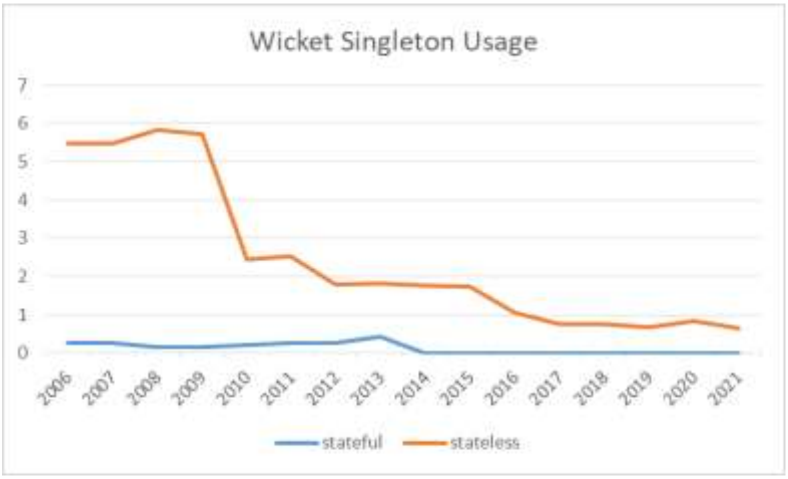

Figure 5.2.11.4: Evolution of singleton usage for Wicket

The usage of stateful singletons is very low as only 1 such instance appeared throughout Wicket’s lifetime. It started at 0.27% and slowly grew to 0.44% right before the respective singleton was removed. For stateless singletons the usage was 5.48% for the first version and it increased to a maximum of 5.86% in October 2007. From there on it began to decrease, more abruptly at first and slowly since 2011. This is due to the fact that the number of stateless singletons decreased, numerous other production classes were created, and they were not singleton clients. For the latest version of Wicket stateless singleton usage is only 0.65%.

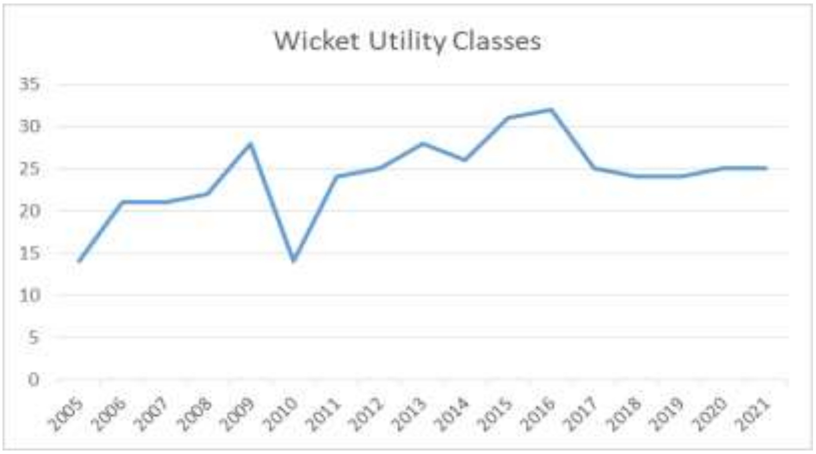

Figure 5.2.11.5: Evolution of utility classes for Wicket

It can be observed that the number of utility classes fluctuates throughout the project's lifespan. In the first years of development it doubled from 14 in 2005 to 28 in 2009. Then the amount of instances dropped to 14 again in 2010; the system was completely refactored during that time period with more than 200 production classes getting removed. From there on it started to slowly increase, thus reaching a peak value of 32 instances in March 2016. Oddly enough, the number of utility classes decreased once again in the next year even though the number of production classes remained roughly the same. Since then it stays almost constant (around 25 instances) until the last version analysed.

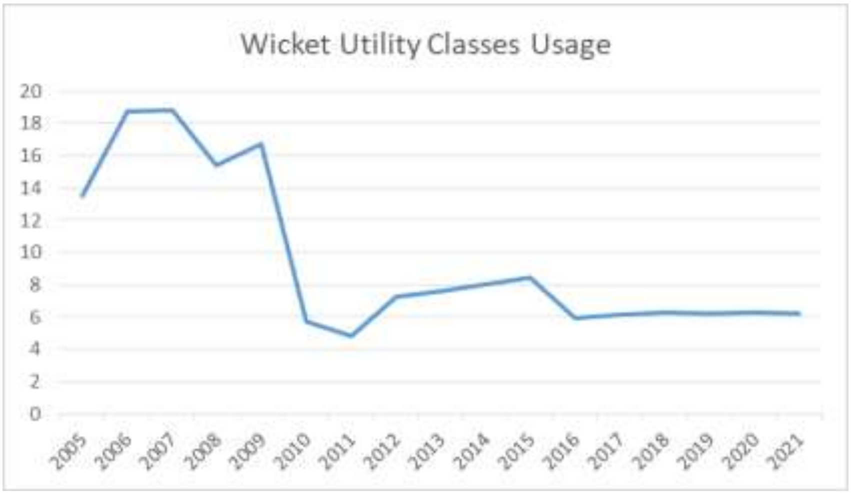

Figure 5.2.11.6: Evolution of utility class usage for Wicket

For the beginning of the development process the usage of utility classes follows the same pattern as the number of instances. However, the drop that occurred in 2010 was significantly steeper; the usage went from the peak value of 16.72% to 5.7% in less than 1 year. Then it fluctuated around 6-8% for the rest of the time period. In the latest version that was studied the usage of utility classes is 6.23%.

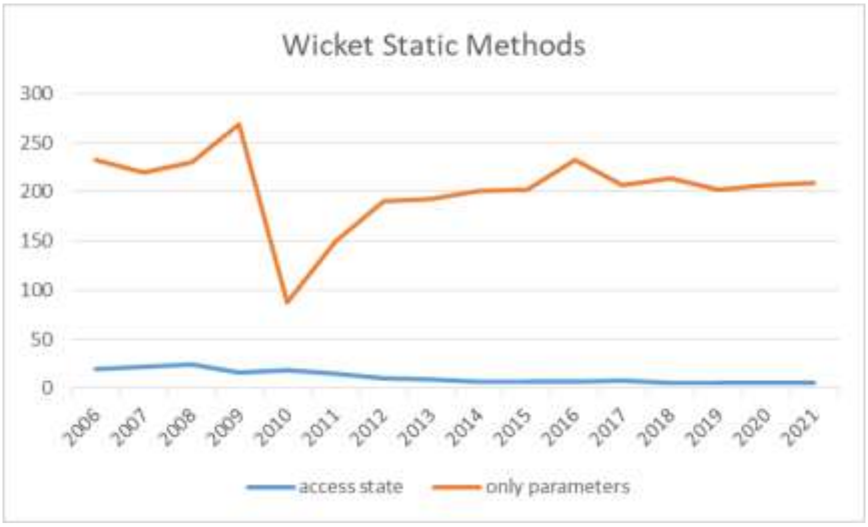

Figure 5.2.11.7: Evolution of static methods for Wicket

The number of static methods that access state increased in the first 2 years of development until reaching a maximum of 24 (0.33%) in 2008. From there on, it decreased continuously throughout the lifespan of Wicket; there are only 6 (0.07%) such instances in the latest version studied. The amount of static methods that solely operate on parameters also increased at first; a maximum value of 269 (3.18%) was encountered in October 2009. Then this value dropped to 87 (1.44%) the following year and started to increase afterwards. There are 209 (2.28%) static methods of this kind in the last version of Wicket.

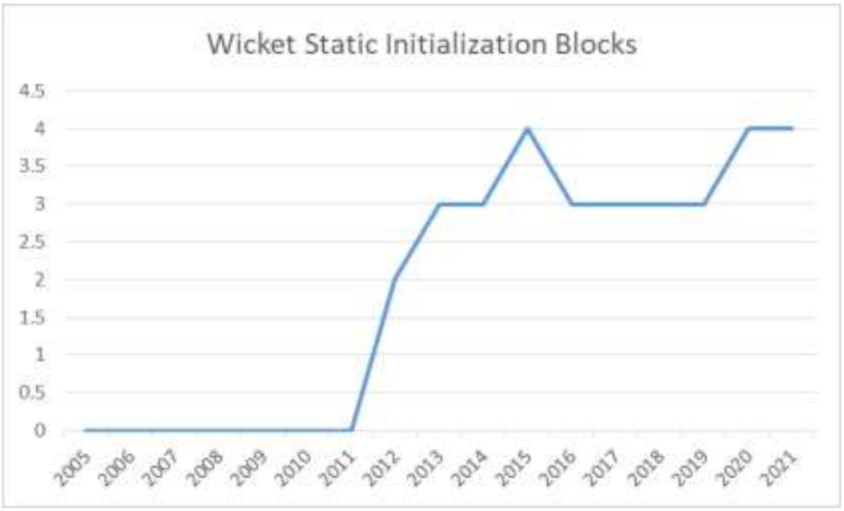

Figure 5.2.11.8: Evolution of static initialization blocks for Wicket

No static initialization blocks were found for Wicket until 2012 when 2 such instances were added as part of the XMLTokener and TagUtils classes. In May 2013 another initialization block appeared in WicketTagIdentifier; a second instance was added to the aforementioned class in November 2015. Approximately 1 year later XMLTokener was removed from the system. Finally, in December 2020 another class that contained a static initialization block, JavaSerializer, was created.

5.3. Impact on class testability

As explained in the previous chapter, we rely on testability scores to compare the classes that contain static constructs to other classes which are similar to them in terms of size and complexity. We do this for each category of static constructs; the comparison is performed both from a quantitative and a qualitative perspective.

5.3.1. BCEL

BCEL is a project that appears to be average in terms of the quantity and quality of its unit tests. However, the classes that contain static non-final attributes have a lower overall testability compared to other similar classes (2.25 vs. 2.5565). They are covered by fewer unit tests (average quantitative score of 2.4667 vs. 2.871) and the tests are of a lesser quality (average qualitative score of 2.0333 vs. 2.2419). Constants on the other hand are much more thoroughly tested (2.9483 vs. 2.5233), but the qualitative score is again lower (2.3631 vs. 2.4685); nevertheless, the overall testability score of the classes that have this kind of static constructs is higher (2.6552 vs. 2.4959) than for similar classes.

Table 5.3.1: Testability of classes with static constructs vs. similar classes for BCEL

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	2.4667	2.0333	2.25	2.871	2.2419	2.5565
	Constants	2.9483	2.3621	2.6552	2.5233	2.4685	2.4959
Singletons	Stateful	3.5	2	2.75	3.5143	2.1429	2.8281
	Stateless	0	-	-	2.9167	2.1538	2.5353
Static methods	Utility classes	3.1818	2.3636	2.7727	3.2353	2.1618	2.6986
	Access state	2.5	2.125	2.3125	2.2874	2.2986	2.293
	Operate on parameters	2.5294	2.4118	2.4706	2.3151	2.3288	2.322
Static init blocks		2.9	2.3	2.6	2.7407	2.1176	2.4292

One stateful singleton (Type) and 2 stateless ones (LONG_Upper and DOUBLE_Upper) were found in the latest version of BCEL studied. The testability of the stateful one is comparable to that of similar classes (2.75 vs. 2.8281). Both its quantitative and its qualitative score are on par with the average scores obtained for the classes that were categorized as similar to it. Surprisingly, the stateless singletons are not addressed by any unit tests (code coverage of 0%), therefore it was impossible to compute a qualitative score and correspondingly an overall testability score. It will be interesting to see how the singletons from the following systems rate in terms of testability.

The utility classes also have average quantitative and qualitive scores that are comparable to the ones of similar classes (3.1818 vs. 3.2353 and 2.3636 vs. 2.1618, respectively); therefore, the overall testability scores are very similar (2.7727 vs. 2.6986). For the rest of the production classes that contain static methods we found that the testability of the ones with methods that access state is comparable to that of similar classes, while for the ones with methods that only operate on parameters it is considerably higher. The latter have higher quantitative (2.5294 vs. 2.3151) and qualitative (2.4118 vs. 2.3288) scores when compared to similar classes. For the former only the score related to quantity is greater (2.5 vs. 2.2874) while the qualitative one is smaller (2.125 vs. 2.2986).

The 5 classes that contain a static initialization block are actually better in terms of testability when compared to other similar classes. Both the average quantitative score (2.9 vs. 2.7407) and the average qualitative one (2.3 vs. 2.1176) are higher, thus the overall testability score is also greater (2.6 vs. 2.4292).

5.3.2. Commons Collections

Table 5.3.2: Testability of classes with static constructs vs. similar classes for Commons Collections

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	-	-	-	-	-	-
	Constants	4.5673	1.5385	3.0529	4.7692	1.6254	3.1923
Singletons	Stateful	-	-	-	-	-	-
	Stateless	4.575	1.675	3.125	4.5897	1.547	3.0683
Static methods	Utility classes	4.5	1.5	3	4.6019	1.5534	3.0777
	Access state	4	2	3	4.6264	1.5287	3.0776
	Operate on parameters	4.7467	1.4133	3.08	4.5534	1.6311	3.0922
Static init blocks		4.5	2	3.25	4.5897	1.547	3.0684

The latest version of Commons Collections does not have any mutable global state instances (stateful singletons and static non-final attributes). There is no significant difference between the average testability score of the classes that contain constants and the corresponding value for other similar classes (3.0529 vs. 3.1923); however, the latter seem to be tested a bit more (average quantitative score of 4.7692 vs. 4.5673) and with better unit tests (average qualitative score of 1.6254 vs. 1.5385).

Similar observations can be made for stateless singletons; the average overall testability of the 5 instances is 3.125, while for similar classes it is 3.0683. The average quantitative scores are almost the same (4.575 vs. 4.5897), but for quality there is a small difference in favour of the first (1.675 vs. 1.547).

For utility classes the average quantitative and qualitative scores are also comparable to the ones obtained for similar classes (4.5 vs. 4.6019 and 1.5 vs. 1.5534, respectively). The latter are a bit higher, thereby causing the overall testability score to also be greater. With regard to the other classes that have static methods, their testability is close to that of similar classes. The classes that contain static methods which access state are covered by fewer tests (average quantitative score of 4 vs. 4.6264), but the tests are of a higher quality (qualitative score of 2 vs. 1.5287); therefore, their overall testability score is a bit lower than the one obtained for classes which are similar to them in terms of size and complexity (3 vs. 3.0776). For the other category however the score is almost the same as for similar classes (3.08 vs. 3.0922); both the quantitative and the qualitative scores are close (4.7467 vs. 4.5534 and 1.4133 vs. 1.6311, respectively).

Finally, the only class that contains a static initialization block, FunctorException, has a higher testability score than the average obtained for the classes that are similar to it. While there is little difference between the quantitative score (4.5 vs. 4.5897), the one for quality is greater (2 vs. 1.547).

5.3.3. Commons Lang

Table 5.3.3: Testability of classes with static constructs vs. similar classes for Commons Lang

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	4.5	1	2.75	4.7414	1.1034	2.9224
	Constants	4.8247	1.0928	2.9588	4.3158	1.1579	2.7368
Singletons	Stateful	-	-	-	-	-	-
	Stateless	5	1.5	3.25	4.7391	1.0957	2.9174
Static methods	Utility classes	4.5714	1.1667	2.869	4.8378	1.0676	2.9527
	Access state	5	0.5	2.75	4.64	1.22	2.93
	Operate on parameters	5	1	3	4.6637	1.1239	2.8938
Static init blocks		4.7778	1.4444	3.1111	4.7383	1.0748	2.9065

An interesting observation can be made with regard to the testability of Lang's classes. Most of them are adequately covered by unit tests (more than 75% line and method coverage), but the respective tests suffer in terms of quality (numerous test classes with a large amount of test smells). The only class that has a static non-final attribute, ToStringBuilder, has lower average scores both for quantity (4.5 vs. 4.7414) and quality (1 vs. 1.1034) compared to other classes that are similar to it; therefore, its overall testability score is also lower. On the other hand, classes that only contain constants do not appear to be tested less / with tests of a lower quality. In fact, the average score for quantity is actually higher (4.8247 vs. 4.3158), while the one for quality is roughly the same as the average obtained for similar classes (1.0928 vs. 1.1579). This causes the overall testability score to be higher (2.9588 vs. 2.7368), albeit not by much.

In the latest version of Lang there are no stateful singletons and only 1 stateless one. The respective instance, ObjectToStringComparator, has a perfect score in terms of quantity; classes which are similar to it are also extensively covered by unit tests, the average quantitative score for them is

4.7391. In terms of quality, the singleton instance has a better score (1.5 vs. 1.0957) which makes its overall testability score considerably higher than the average one for similar classes (3.25 vs. 2.9174).

For utility classes their average testability score (2.869) is almost equal to the corresponding value for similar classes (2.9527). While they seem to be addressed by fewer tests (average quantitative score of 4.5714 vs. 4.8378), the corresponding test are of a better quality (average qualitative score of 1.1667 vs. 1.0676). From the rest of the production classes that contain static methods, those with methods that access state have a lower testability compared to other similar classes, while for the ones with static methods that only operate on parameters the testability score is higher (3 vs. 2.8938). For the classes from the first category, the average quantitative score is greater (5 vs. 4.64) while the quality score is considerably lower (0.5 vs. 1.22); this causes their overall testability score to be lower as well (2.75 vs. 2.93), albeit not by much. Different observations can be made with regard to the classes that contain static methods that solely operate on parameters. Their average quantitative score is also higher (5 vs. 4.6637), but the qualitative score is close to that of similar classes (1 vs. 1.1239) which makes the overall testability score higher.

Finally, the classes with static initialization blocks have, on average, a higher testability score than other classes which are similar to them (3.1111 vs. 2.9065). The average scores are roughly the same for quantity (4.7778 vs. 4.7383), but for quality there seem to be fewer smells in their corresponding unit tests (1.4444 vs. 1.0748).

5.3.4. Commons Math

Table 5.3.4: Testability of classes with static constructs vs. similar classes for Commons Math

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	4.1667	0.3333	2.25	4.6129	0.4996	2.5563
	Constants	4.493	0.4406	2.4668	4.6066	0.4918	2.5492
Singletons	Stateful	-	-	-	-	-	-
	Stateless	4	0.5	2.25	4.413	0.4496	2.4313
Static methods	Utility classes	4.4444	0.5556	2.5	4.5167	0.4438	2.4802
	Access state	4.4375	0.75	2.5938	4.4841	0.7841	2.6341
	Operate on parameters	4.2174	0.6304	2.4239	4.2294	0.4745	2.352
Static init blocks		3.875	0.625	2.25	4.5219	0.4548	2.4883

For Math the average qualitative scores are very low both for classes with instances of static constructs and for the groups of similar classes. The biggest difference in terms of testability was observed for the classes that contain static non-final attributes, namely DfpField, DSCompiler, and GeneticAlgorithm. Their average quantitative score is 4.1667 (compared to 4.6129 for similar classes), while the one for quality is 0.3333 (vs. 0.4996); thus, there is a 0.3063 difference between their overall testability score and the corresponding value for the classes which are considered similar to them. This is the largest difference encountered from all the categories of static constructs, thereby suggesting that these types of instances have the highest impact on class testability. For constants the values obtained are quite similar (4.493 vs. 4.6066 for quantity and 0.4406 vs. 0.4918 for quality); there is little difference between the overall testability scores (2.4668 vs. 2.5492) for this kind of instances compared to other similar classes.

There is only 1 stateless singleton in the version studied, Decimal64Field. While its quantitative score is smaller than the average one obtained for classes which are similar to it (4 vs. 4.413), the quantity score is a bit higher (0.5 vs. 0.4496). This makes their overall testability scores comparable (2.25 vs. 2.4313). For utility classes the respective scores are almost identical (2.5 vs. 2.4802); the average quantitative score is a bit lower (4.4444 vs. 4.5167), while the qualitative one is higher (0.5556 vs. 0.4438).

For the rest of the production classes that contain static methods, their overall testability score is comparable the one obtained for other classes. The ones with methods that access state have a slightly lower score (2.5938 vs. 2.6341), while for the classes with static methods that only operate on parameters the value is a bit higher (2.4239 vs. 2.352) than for similar classes. Those from the first category are tested less (4.4375 vs. 4.4841) and with tests of a lower quality (0.75 vs. 0.7841). For the latter only the qualitative score is larger (0.6304 vs. 0.4745), the quantitative one is roughly the same (4.2174 vs. 4.2294).

The overall testability of the 10 classes with static initialization blocks seems to be a bit lower than for other similar classes (2.25 vs. 2.4883). This is due to the fact that the average quantitative score is significantly lower (3.875 vs. 4.5219); on the other hand, the average qualitative score is higher (0.625 vs. 4548).

5.3.5. Digester

Table 5.3.5: Testability of classes with static constructs vs. similar classes for Digester

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	-	-	-	-	-	-
	Constants	3.4583	3.375	3.4167	3.582	3.1214	3.3517
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	4	3	3.5	3.6607	3.2105	3.4356
	Access state	-	-	-	-	-	-
	Operate on parameters	3.6667	3.1667	3.4167	3.4776	3.3039	3.3908
Static init blocks		-	-	-	-	-	-

As seen in the first section of this chapter, there are very few instances of static constructs in this system. However, the quantitative and qualitative scores are generally high for Digester (over 3.5 and between 3-3.5, respectively). For the 24 classes with constants the overall testability score is almost the same as for similar classes (3.4167 vs. 3.3517). They are addressed by fewer unit tests (average quantitative score of 3.4583 vs. 3.582), but the tests are of a better quality (qualitative score of 3.375 vs. 3.1214).

For the 2 utility classes the situation is the other way around. They have a higher average quantitative score (4 vs. 3.6607) and a lower qualitative score (3 vs. 3.2105); nevertheless the outcome is the same, their overall testability score is greater than that of similar classes (3.5 vs. 3.4356). Additionally, there are 6 production classes that contain static methods that solely operate on parameters. For them the situation is exactly the same as for utility classes; their average quantitative score is higher (3.6667 vs. 3.4776) and the one for quality is lower (3.1667 vs. 3.3039) than for similar classes. Finally, the overall testability scores are more or less the same (3.4167 vs. 3.3908).

5.3.6. Geode

The testability scores for Geode are very low, even the quantitative ones. Classes that contain static non-final attributes have a lower overall score (1.0373 vs. 1.1628) compared to other similar classes, while for the ones with constants the values are comparable (1.1339 vs. 1.0884). The former are covered by fewer unit tests (quantitative score of 1.2521 vs. 1.3492) and the respective tests are of a lower quality (qualitative score of 0.8224 vs. 0.9763). For the latter the quantity is almost the same (1.3084 vs. 1.2929), but the quality is better (0.9593 vs. 0.8893).

Table 5.3.6: Testability of classes with static constructs vs. similar classes for Geode

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	1.2521	0.8224	1.0373	1.3492	0.9763	1.1628
	Constants	1.3084	0.9593	1.1339	1.2929	0.8839	1.0884
Singletons	Stateful	1.2308	0.9231	1.077	1.2882	0.9176	1.1029
	Stateless	1.7222	0.8889	1.3056	1.4831	0.9487	1.2159
Static methods	Utility classes	1.6599	0.9594	1.3097	1.2586	0.9122	1.0854
	Access state	1.0751	1.0491	1.0621	1.3332	0.807	1.0701
	Operate on parameters	1.6201	0.8939	1.257	1.2655	0.9947	1.1301
Static init blocks		1.8026	0.9211	1.3619	1.3759	1.0674	1.2217

Similar observations can be made with regard to the 2 types of singletons. The stateful ones have a lower quantitative score (1.2308 vs. 1.2882) and an almost identical qualitative score (0.9231 vs. 0.9176) when compared to similar classes; therefore, their overall testability is also a bit lower (1.077 vs. 1.1029). The stateless singletons are addressed by considerably more tests (1.7222 vs. 1.4831 for quantity) which are more or less the same in terms of quality (0.8889 vs. 0.9487). The overall testability score for these instances is higher (1.3056 vs. 1.2159) than the corresponding value for similar classes.

For utility classes both the quantitative and the qualitative scores are greater (1.6599 vs. 1.2586 and 0.9594 vs. 0.9122, respectively). This makes their overall testability score significantly higher (1.3097 vs. 1.0854) than the one obtained for classes that are similar to them in terms of size and complexity. For the rest of the classes that contain static methods, we found that those with methods that access their state are just as testable as other similar classes, while the ones with static methods that solely operate on parameters have an even higher testability. For the ones from the first category the coverage is lower (1.0751 vs. 1.3332), but the quality of the tests is better (1.0491 vs. 0.807). It is the other way around for the classes from the second category; they have a much higher quantitative score (1.6201 vs. 1.2665) compared to similar classes, but the qualitative score is a bit smaller (0.8939 vs. 0.9947).

Finally, the classes with static initialization blocks have a better overall testability score (1.3619 vs. 1.2217) than other production classes. Their quantitative score (1.8026) is the highest one encountered for classes with static constructs, while their qualitative score is also good (0.9211).

5.3.7. jHotDraw

For jHotDraw it was impossible to investigate the impact of the different types of static constructs on class testability. As explained in Chapter 4, this system was included in the study due to the fact that it was used as reference in an article that discusses variations of the Singleton design pattern. However, the amount of testing performed on the latest version of the system is insufficient for a proper analysis on class testability; the test / production code ratio is 0.05 and there are only 200 unit test compared to 2713 production methods. Selecting this project for the empirical study was needed especially for the analysis on the evolution of singletons. As an example, we checked if the number of instances encountered at certain moments in time is in accordance with what is discussed in the article. The system will also be useful in our analyses on change- / defect-proneness that follow.

5.3.8. Pig

With regard to static attributes, there is a big difference between the testability of the classes that contain non-final ones and those with constants. The overall testability score of the later is almost identical (1.7206 vs. 1.7) to that of similar classes; even the quantitative and qualitative scores are roughly the same (2.5588 vs. 2.5333 and 0.8824 vs. 0.8667, respectively). In contrast, the classes with static non-final attributes are covered by fewer unit tests (average quantitative score of 2.0909 vs. 2.4739) and the respective tests are of a poorer quality (qualitative score of 0.8182 vs 0.9211).

Table 5.3.8: Testability of classes with static constructs vs. similar classes for Pig

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	2.0909	0.8182	1.4546	2.4739	0.9211	1.6975
	Constants	2.5588	0.8824	1.7206	2.5333	0.8667	1.7
Singletons	Stateful	2.2857	0.8571	1.5714	2.5636	0.8916	1.7276
	Stateless	2.5	1	1.75	2.4889	0.8491	1.669
Static methods	Utility classes	2.4333	0.9167	1.675	2.3261	0.913	1.6196
	Access state	2.2727	0.9091	1.5909	2.5526	1.0263	1.7895
	Operate on parameters	2.4375	0.9375	1.6875	2.2	0.9556	1.5778
Static init blocks		2.3571	0.7857	1.5714	2.2708	0.9375	1.6042

The testability of singletons varies depending on the type of the instances. The 14 stateful ones are tested less thoroughly (average quantitative score of 2.2857 vs. 2.5636) compared to similar classes, but the quality of the unit tests is roughly the same (qualitative score of 0.8571 vs. 0.8916); all in all, the overall testability score is lower for this kind of singletons (1.5714 vs. 1.7276). For the 4 stateless ones both the values for quantity and quality are higher (2.5 vs. 2.4889 and 1 vs. 0.8491). This makes the overall testability score greater as well (1.75 vs. 1.669), but all 3 scores are very close.

For utility classes both averages are higher (2.4333 vs. 2.3261 for quantity and 0.9167 vs. 0.913 for quality), although not by much; thus the overall testability score is also greater (1.675 vs. 1.6196). On the other hand, production classes that have static methods which access state are tested less (2.2727 vs. 2.5526) and with unit tests that are worst in terms of quality (0.9091 vs. 1.0263). This suggests that they are less testable than classes that are similar to them in terms of size and complexity. For classes with static methods that only operate on parameters the situation resembles the one for utility classes. The difference between the average quantitative scores is slightly bigger (2.4375 vs. 2.2), but the qualitative score is more or less the same (0.9375 vs. 0.9556). In consequence, the overall testability score is a bit higher (1.6875 vs. 1.5778).

Classes with static initialization blocks have a similar testability as other production classes (overall score of 1.5714 vs. 1.6042). Both the quantitative and qualitative scores are comparable to those of similar classes (2.3571 vs. 2.2708 and 0.7857 vs. 0.9375, respectively).

5.3.9. Spring Core

Table 5.3.9: Testability of classes with static constructs vs. similar classes for Spring Core

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	3.3	3.05	3.175	3.9356	3.374	3.6548
	Constants	3.8293	3.2439	3.5366	3.8696	3.4348	3.6522
Singletons	Stateful	-	-	-	-	-	-
	Stateless	4.2	4	4.1	3.8239	3.5535	3.6887
Static methods	Utility classes	3.697	3.4242	3.5606	3.8761	3.2301	3.5531
	Access state	3.5769	3.3077	3.4423	3.9593	3.3667	3.663
	Operate on parameters	4.3125	3.875	4.0938	3.7786	3.1985	3.4885
Static init blocks		4	3.2	3.6	3.8235	3.2794	3.5515

There are 10 classes that contain static non-final attributes and their average testability is considerably lower than the corresponding value for similar classes (3.175 vs. 3.6548); this is by far the biggest difference encountered compared to the other types of static constructs. Both the average quantitative and qualitative scores are lower (3.3 vs. 3.9356 and 3.05 vs. 3.374, respectively) than the ones obtained for similar classes. On the other hand, for classes with constants these differences are much smaller (3.8293 vs. 3.8696 for quantity and 3.2439 vs. 3.4348 for quality). The average overall testability scores are comparable, 3.5366 for the classes that contain constants and 3.6522 for other classes that were categorized as similar to them.

There are no stateful singletons in the version of Spring Core that was investigated. Surprisingly, for the 5 stateless singletons found, their testability appears to be higher than that of similar classes both in terms of quantity (4.2 vs. 3.8239) and quality (4 vs. 3.5535). This causes the overall testability score to be significantly higher (4.1 vs. 3.6887), thereby suggesting that this kind of singletons are not more difficult to test.

For utility classes their testability is comparable to that of similar classes (3.5606 vs. 3.5531). Although they are covered by fewer unit tests (average quantitative score of 3.697 vs. 3.8761), these tests are of a better quality (average qualitative score of 3.4242 vs. 3.2301). The testability of the remaining production classes that contain static methods is different depending on the type of the methods. Those with static methods that access their state scored lower for both quantity and quality (3.5769 vs. 3.9593 and 3.3077 vs. 3.3667, respectively). For the ones with static methods that solely operate on parameters the corresponding values are substantially greater (4.3125 vs. 3.7786 and 3.875 vs 3.1985) than for similar classes. This proves that the former are more difficult to test, while the latter are highly testable.

Finally, for the 25 classes with static initialization blocks the situation is the opposite of the one encountered for utility classes. The quantitative score is higher (4 vs. 3.8235) and the qualitative one lower (3.2 vs. 3.2794), thus making the overall testability scores very similar (3.6 vs. 3.5515).

5.3.11. Wicket

Table 5.3.11: Testability of classes with static constructs vs. similar classes for Wicket

Category		**Instances**			**Similar classes**		
		Avg. Sc. Quant.	**Avg. Sc. Qual.**	**Avg. Sc. Test.**	**Avg. Sc. Quant.**	**Avg. Sc. Qual.**	**Avg. Sc. Test.**
Static attributes	Non-final	2.8667	2.0667	2.4667	2.9363	2.5247	2.7305
	Constants	2.9394	2.4061	2.6728	2.7556	2.6222	2.6889
Singletons	Stateful	-	-	-	-	-	-
	Stateless	2.875	2.625	2.75	2.7323	2.5743	2.6533
Static methods	Utility classes	3.1875	2.625	2.9063	2.8168	2.4059	2.6114
	Access state	2.875	2.375	2.625	3.1089	2.3932	2.7511
	Operate on parameters	3.075	2.5	2.7875	2.7586	2.3809	2.5698
Static init blocks		3.1	2.6	2.85	3.2105	2.5694	2.89

Wicket is a project in which the overall quantitative and qualitative scores are very similar, with a small difference in favour of the first. The classes that contain static non-final attributes seem to be tested a bit less (average quantitative score of 2.8667 vs. 2.9363) and with unit tests of a lower quality (average qualitative score of 2.0667 vs. 2.5247) compared to other similar classes; therefore, their overall testability score is significantly smaller (2.4667 vs. 2.7305). On the other hand, the score for classes with constants is almost identical to that of similar classes (2.6726 vs. 2.6889). Their average quantitative score is higher (2.9394 vs. 2.7556), while the qualitative score is lower (2.4061 vs. 2.6222).

There are no stateful singletons in the latest version of Wicket. For the 8 stateless ones both the average quantitative and qualitative scores are higher (2.875 vs. 2.7323 and 2.625 vs. 2.5743, respectively), but not by a large margin; thus, the overall testability score is also greater (2.75 vs 2.6533) than for similar classes.

Utility classes actually have the highest testability score (2.9063) out of all the static constructs investigated. They are addressed by more tests (3.1875 vs. 2.8168) compared to classes that are similar to them in terms of size and complexity; the quality of the unit tests is also better (2.625 vs. 2.4059). For the other classes that contain static methods, we found that those with methods that access their state have a lower testability than other similar classes (2.625 vs. 2.7511), while for the ones with static methods that only operate on parameters the testability is considerably higher (2.7875 vs. 2.5698). For the former, although the quality of the tests is roughly the same (2.375 vs. 2.3932) as for similar classes, their quantity is much lower (2.875 vs. 3.1089). For the second category, the average scores for both quantity and quality are substantially higher (3.075 vs. 2.7686 and 2.5 vs. 2.3809, respectively).

Finally, for classes with static initialization blocks there does not appear to be any difference in terms of testability when compared to similar classes. The average quantitative score is slightly lower (3.1 vs. 3.2105), while the corresponding value for quality is a bit higher (2.6 vs. 2.5694). This causes the overall testability scores to be almost the same (2.85 vs. 2.89).

5.4. Impact on change- / defect-proneness

The proposed procedures for quantifying change- and defect-proneness were explained in Chapter 3. They are used to determine whether or not the classes with different types of static constructs were modified more frequently / more fine-grained source code changes were performed on them. The only difference between them is that for error-proneness we only take into account the commits that were categorize as bug-fixes. The following table contains an overview of the bug-fix commits identified for each project.

Table 5.4 Bug-fix commits identified

System	**Total # Jira bugs**	**# Commits containing issue keys**	**# Commits with issue keys corresponding to bugs**	**# Commits identified based on keywords**	**Total # bug-fix commits**
BCEL	252	195	110	141	251
Collections	362	485	175	373	548
Commons Lang	707	1466	571	394	965
Commons Math	728	2030	667	646	1313
Digester	122	82	38	381	419
Geode	4990	8877	4349	1075	5424
jHotDraw	-	-	-	76	76
Pig	3109	3368	2094	150	2244
Spring Core	-	-	-	1903	1903
Tomcat	-	-	-	4696	4696
Wicket	4163	6573	3727	3161	6888

It can be observed that for 3 of the 11 systems we were not able to find a corresponding Jira issue tracker; therefore, for these projects we detected bug-fix commits solely based on keywords. Our evaluation of defect-proneness for the 3 systems might not be as accurate as for the others. In both assessments (change- and error-proneness) we compared the average number of changes performed on classes that have a certain type of static construct with the corresponding value for classes which are similar to them in terms of size and complexity. Furthermore, we also computed the average number of modifications per commit in order to determine if the instances of interest were altered in more commits than the similar classes. Finally, we were keen to observe which types of fine-grained source code changes occurred the most frequently and to establish whether or not the rankings are different for various categories of static constructs / similar classes. The acronyms correspond to the change types presented in Table 3.2.

5.4.1. BCEL

Table 5.4.1.1: Change-proneness of classes with static constructs vs. similar classes for BCEL

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	200.0667	7.7333	24.8584 SD 22.859 SU 19.3269 SI 7.1643 SPC 3.4655 DU	40.6875	3.1875	27.1689 SU 11.5408 SI 10.9397 SD 7.0193 IAC 5.57 SPC
	Constants	77.2459	5.9836	19.7646 SU 12.9013 SD 11.602 SI 8.1474 IAC 7.7499 RAT	35.5031	2.8696	30.2747 SU 13.5497 SI 13.471 SD 5.6858 SPC 4.6186 IAC
Singletons	Stateful	85	2.931	23.5294 SU 17.6471 RF 12.9412 SI 8.2353 SD 4.7059 PTC	46.8298	3.3665	26.4632 SU 14.2875 SD 11.8403 SI 5.9142 IAC 5.8528 SPC
	Stateless	8.5	1.3077	17.6471 SU 11.7647 AAM 11.7647 AOS 11.7647 ROS 11.7647 SD	37.1312	3.378	26.4576 SU 13.265 SD 12.8418 SI 6.9082 IAC 4.8417 SPC
Static methods	Utility classes	135.9	6.4	19.1317 SI 17.0714 SU 16.1884 SD 8.1678 DU 6.0338 SPC	44.5442	3.2841	27.2164 SU 13.0244 SD 12.3262 SI 6.2413 IAC 5.82 SPC
	Access state	148.375	6.125	24.4082 SD 23.2566 SU 19.0019 SI 7.4216 SPC 3.4869 RF	41.3593	3.1866	28.1215 SU 12.5436 SI 10.9173 SD 7.0649 IAC 5.5024 SPC
	Operate on parameters	142.8824	5.6471	29.4772 SU 16.056 SI 13.7505 SD 13.2977 SPC 5.4755 CEC	42.4727	3.2596	25.9762 SU 12.3384 SI 12.1875 SD 6.7739 IAC 4.6703 SPC
Static init blocks		132.8	4.8	19.5783 SU 16.8675 SD 15.0602 SI 6.9277 RMO 6.4759 SPC	45.7937	3.3466	26.7129 SU 13.1254 SD 12.7556 SI 6.1236 IAC 5.8117 SPC

The 15 production classes that have static non-final attributes are more change-prone compared to similar classes. The average number of modifications is roughly 5 times higher (200.0667 vs.

40.6875) while the number of changes per commit is less than 2.5 times higher (7.7333 vs. 3.1875). The top 3 change types are the same, but the following 2 and the percentages are very different.

Although there is a difference between the classes with constants and other similar classes in terms of average number of changes (77.2459 vs. 35.5031) and number of modifications per commit (5.9836 vs. 2.8696), this difference is not nearly as great as for the classes that contain static non-final attributes. Four of the top 5 change types are the same, albeit the percentages are fairly different.

The statefull singleton (Type) is more change-prone compared to the classes that were categorized as similar to it. It suffered 85 fine-grained modifications during 29 commits; for similar classes the average number of modifications is almost half (46.8298) and the number of changes per commit is a bit higher (3.3665 vs. 2.913). The corresponding values for the 2 stateless singletons are much lower; their average number of changes is 8.5 while the amount of modifications per commit is 1.3077, thus indicating that they are less change-prone. Regarding the top 5 change types, only statement updates are deletions appear in all the ranking; the results might be inconclusive due to the very small number of instances (1 and 2, respectively).

The change-proneness of utility classes is much higher than for other similar classes. The average number of changes is triple (135.9 vs. 44.5442) and there are twice as many modifications per commit (6.4 vs. 3.2841). The other production classes that contain static methods also appear to be more change-prone compared to similar classes. Both the ones with static methods that access state and those with methods that only operate on parameters have had, on average, a higher number of modifications performed on them (148.375 vs. 41.3593 and 142.8824 vs. 42.4727, respectively). The number of changes per commit is roughly double than for similar classes in both cases (6.125 vs. 3.1866 and 5.6471 vs. 3.2586). Finally, 4 of the top 5 change types are the same, though their percentages and order are quite different.

Utility (448 changes) also contains a static initialization block along with 4 other classes; the average number of changes for these instances is considerably higher than for similar classes (132.8 vs. 45.7937) and there are also more changes per commit (4.8 vs. 3.3466). However, they are not very different with regard to the types of changes that were performed; the top 3 changes types are the same and even the percentages are quite close.

Table 5.4.1.2: Defect-proneness of classes with static constructs vs. similar classes for BCEL

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	52.2667	12.8	53.5714 SD 13.648 SI 8.9286 SU 5.4847 SPC 3.9541 CEC	13.75	4.2012	20.4878 SD 16.7627 SU 15.2993 SI 14.0133 DD 5.9424 DU
	Constants	26.5278	8.1111	24.6388 SD 22.7376 DD 13.6122 SI 7.3004 SU 6.5399 SPC	12.0559	4.1189	32.3666 SD 20.4176 SI 15.8353 SU 5.6265 DU 5.3364 RF
Singletons	Stateful	17	3.4	47.0588 RF 17.6471 DAC 11.7647 SU 11.7647 AAM 11.7647 IAC	16.9775	4.9326	29.186 SD 14.957 SI 14.7584 SU 10.4897 DD 4.9301 DU
	Stateless	1	1	100 SU	17.6	4.8	32.5875 SD 17.7281 DD 9.4664 SI 7.9174 DU

							7.1715 SU
Static methods	Utility classes	39.6	9	22.2222 SI 17.1717 SOC 12.6262 SD 12.1212 DU 10.606 SPC	16.3276	4.8046	30.1654 SD 15.3819 SU 14.3611 SI 11.158 DD 4.3999 DU
	Access state	71.5	4	54.5455 SD 13.7004 SI 8.1946 SU 4.2254 SPC 3.073 RF	14.2013	4.5849	20.1949 SD 17.0062 SU 15.279 SI 13.9061 DD 6.023 DU
	Operate on parameters	12.375	3.0625	20.7071 SU 19.697 SD 16.1616 SI 7.0707 RF 6.5657 SPC	17.4294	5.1043	28.6727 SD 14.7835 SI 14.3259 SU 11.1228 DD 4.8222 DU
Static init blocks		39	7.75	17.9487 SI 12.8205 SD 10.8974 SOC 9.6154 SPC 8.9744 SU	16.4743	4.8571	29.8994 SD 15.0538 SU 14.7069 SI 10.9955 DD 4.8907 DU

All 15 production classes that contain static non-final attributes were modified during bug-fix commits. Same as for change-proneness, these classes are more error-prone compared to other similar classes; the average number of modifications is almost 4 times higher (52.2667 vs. 12.8) while the number of changes per commit is 3 times higher (12.8 vs. 4.2012). An interesting observation is that more than half of the changes performed on the classes with static non-final attributes are statement deletions; for the other classes this percentage is a bit over 20%.

Classes with constants are not as error-prone as they are change-prone. Although the average number of changes for such classes is more than double compared to other similar classes (26.5278 vs. 12.0559), the amount of modifications per commit is also double (8.1111 vs. 4.1189); this implies that the number of bug-fix commits in which these classes were altered is roughly the same as for similar classes. The top 5 change types are quite different, both regarding the types and especially the percentages.

The stateful singleton was modified in 5 bug-fix commits. The average number of changes is almost identical (17 vs 16.9775) to the one obtained for similar classes; however, there are fewer modifications per commit (3.4 vs. 4.9326), thereby implying that the singleton was altered more frequently. This is not the case for the 2 stateless singletons; each of them suffered only 1 fine-grained source code change (in separate bug-fix commits). The top 5 change types are also very different; for example, the 2 modifications performed on the stateless variants were statement updates.

Five utility classes were modified during bug-fix commits and the average number of changes is higher than for similar classes (39.6 vs. 16.3276). However, the ratio is considerably smaller than the one obtained for change-proneness. The number of changes per commit is also greater (9 vs. 4.8046). There are significant differences between the top 5 change types for utility classes and other similar classes, both in terms of order and percentage-wise. Unlike for change-proneness, only the classes with static methods that access state have a higher error-proneness than similar classes; those with static methods that only operate on parameters do not. For the first category, the average number of changes is 5 times higher (71.5 vs. 14.2013) while the amount of modifications per commit is comparable (4 vs. 4.5849); this indicates that there are more bug-fix commits in which they were altered. This is not the case for classes with methods that solely operate on parameters; they actually appear to be less error-prone when compared to similar classes (12.375 vs. 17.4294 average number of modifications and 3.0625 vs. 5.1043 number of changes per commit). In terms of top 5 change types, the 3 statement-level changes were the most commonly occurring in both cases while the following 2 differ when compared to those of similar classes.

The situation for the classes with static initialization blocks is similar to that of utility classes. They suffered more changes on average (39 vs. 16.4743) and the number of modifications per commit is higher (7.75 vs. 4.8571). In terms of top change types, they are different than the ones encountered for similar classes; for example, less that 13% of the modifications are statement deletes for the classes that contain static initialization blocks, while for the other classes the corresponding value is roughly 30%.

5.4.2. Commons Collections

Table 5.4.2.1: Change-proneness of classes with static constructs vs. similar classes for Commons Collections

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	-	-	-	-	-	-
	Constants	87.7215	6.0823	19.1631 SU 13.456 DU 10.8658 SI 9.8413 SD 7.0851 PTC	74.1667	6.9667	19.457 SU 13.4232 DU 11.0412 SI 8.7341 SD 6.8165 PTC
Singletons	Stateful	-	-	-	-	-	-
	Stateless	9	2.6667	51.8519 SU 14.8148 DD 7.4074 ROS 3.7037 AF 3.7037 ATC	68.7517	5.9139	17.5522 SU 14.7263 DU 11.6602 SI 10.541 SD 5.7717 PTC
Static methods	Utility classes	181.3181	6.4091	22.2361 DU 17.1221 SU 12.5094 SI 8.6489 PTC 7.8967 SD	73.2124	6.4027	19.757 SU 11.326 DU 10.5403 SI 9.8634 SD 6.5999 PTC
	Access state	77	5.1333	19.2941 DU 17.1765 SU 16 SI 7.2941 DD 6.5882 API	68.8007	5.9088	18.6251 SU 12.5969 DU 11.5541 SI 10.6489 SD 7.8795 PTC
	Operate on parameters	55.6723	4.8655	19.4717 SU 13.4038 DU 9.2679 SI 9.1623 SD 7.7736 SPC	76.1559	6.5323	17.1857 SU 14.8581 DU 12.757 SI 9.7 SD 6.2902 PTC
Static init blocks		55	13.75	25.4545 SI 18.1818 SD 14.5455 AF 7.2727 AF 7.2727 AOS	82.915	6.3765	19.2969 SU 13.4717 DU 10.8838 SI 9.458 SD 7.0166 PTC

There are no classes that contain static non-final attributes. For classes with constants, both the average number of changes (87.7215 vs. 74.1667) and the number of modifications per commit (6.0823 vs. 6.9667) are comparable to those of similar classes. Additionally, the order and percentages for the top 5 change types are nearly identical.

The stateless singletons suffered, on average, 9 fine-grained changes; this value is much smaller than the corresponding one for similar classes (68.7517). The number of modifications per commit is less than half 2.6667 vs. 5.9139); therefore, the singletons were altered in much fewer commits. The changes performed on them are also very different compared to the top 5 change types for similar classes; however, statement updates were first in the rankings in both cases.

The utility classes were modified more compared to other similar classes (average number of changes of 181.3181 vs. 73.2124). This is mainly due to 3 utility classes which suffered most of the modifications: CollectionUtils (900 changes), IteratorUtils (470), and MapUtils (545). However, the number of changes per commit are almost identical, thus showing that the utility classes are changed in more commits. Also, the top 5 change types are the same, although the percentages are very different; over 22% of the changes are doc updates for utility classes (compared to 11.326%), proving once again that they are a central part of systems which are structured as libraries. For the rest of the classes that contain static methods, it was found that those with methods that access state are a bit more change-prone that similar classes while the ones with static methods that solely operate on parameters are actually less. The first have a higher average number of changes (77 vs. 68.8007) and roughly the same number of modifications per commit (5.1333 vs. 5.9088). On the other hand, for classes with static methods that only operate on parameters the average is substantially lower (55.6723 vs. 76.1559) than for similar classes and the number of modifications per commit is only slightly lower (4.8655 vs. 6.5323). With regard to the top 5 change types, classes with static methods that access state are one of the few cases in which a statement-level modification, statement deletes, does not appear in the list. This is not the case for those with static methods that only operate on parameters; for this category the ranking is very similar to the one obtained for similar classes (both in terms of order and percentages).

There is only one class that contains a static initialization block, FunctorException, on which 55 changes were performed over 4 commits. Although the number of changes is lower compared to the average for the classes that are similar to it, but the number of modification per commit is higher (13.75 vs. 6.3765). The types of changes and their percentages are very different, but the results might be skewed because there is only 1 instance of interest.

Table 5.4.2.2: Defect-proneness of classes with static constructs vs. similar classes for Commons Collections

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	-	-	-	-	-	-
	Constants	10.9034	3.8759	21.907 DU 13.9558 SD 12.6691 SU 11.9762 SI 7.0934 RF	13.8906	3.875	28.9089 DU 13.3858 SD 10.7987 SU 8.6614 SI 7.9865 DD
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	49.8235	5.2353	28.3353 DU 12.0425 SU 11.4522 PTC 9.5632 SI 8.0283 SD	16.0052	3.7552	22.1608 DU 15.4247 SD 12.3007 SU 11.6824 SI 6.8012 RF
	Access state	21	5.25	29.3103 SI 29.3103 DU 10.3448 API 10.3448 SU 8.6207 CI	15.9386	3.7632	25.1789 DU 13.2361 SD 11.1723 SU 10.732 SI 6.852 RF
	Operate on	11.6293	3.0603	28.9844 DU	20.0256	4.5299	23.0901 DU

	parameters			13.7139 SD 12.4537 SU 11.8606 SI 9.3403 RF			12.6761 SD 10.542 SI 10.414 SU 5.2924 RF
Static init blocks		-	-	-	-	-	-

As mentioned before, there are no production classes with static non-final attributes. For the ones that contain constants we observed that they are not more defect-prone than other similar classes. The average number of modifications are similar (10.9034 vs. 13.8906) and the number of changes per commit are almost identical (3.8759 vs. 3.875). The top 4 change types are the same, but the percentages do differ to some extent.

None of the changes performed on the stateless singletons occurred during bug-fix commits. Therefore, even though we were able to evaluate their change-proneness, the defect-proneness of the singletons could not be assessed.

The average number of changes in bug-fix commits is higher for utility classes compared to similar classes (49.8235 vs. 16.0052). However, the number of changes per commit is comparable (5.2353 vs. 3.7552), thus suggesting that the former were modified in more commits in which errors were repaired. Even though the top 5 change types are roughly the same, the percentages for them vary significantly; only 8% are statement deletes for utility classes while for the others this percentage is around 15.5%. For other classes that contain static methods the situation is similar to the one observed for change-proneness. Those with methods that access state have, on average, a higher number of modifications (21 vs. 15.9386); their number of changes per commit is also a bit higher (5.25 vs. 3.7632). On the other hand, classes with static methods that only operate on parameters have a lower defect-proneness compared to similar classes; their average number of modifications is almost half (11.6293 vs. 20.0256) and the number of changes per commit is lower (3.0603 vs. 4.5299). The top 5 change types are significantly different for the first category both order- and percentage-wise. For the latter category all 5 modification types are exactly the same and the percentages are quite similar.

None of the 4 commits in which FunctorException (the only class that contains a static initialization block) was modified are bug fixes.

5.4.3. Commons Lang

Table 5.4.3.1: Change-proneness of classes with static constructs vs. similar classes for Commons Lang

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	280	8.4848	28.5714 DU 20 SI 14.2857 SU 12.1429 SD 4.6429 RF	203.6	5.9353	22.6395 SD 20.9234 SI 11.4007 SU 10.4646 DU 6.5786 RF
	Constants	84.5472	6.7925	22.5151 SI 20.9535 SD 11.4515 SU 10.1452 DU 6.8497 RF	72.7692	4.5692	23.277 SI 20.2114 SD 13.5729 DU 11.2474 SU 4.7357 RF
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	516.7907	9.1628	24.1427 SI 21.9332 SD 11.1781 SU 10.2871 DU	98.9844	4.8672	

				6.6736 RF			
	Access state	545	11.8839	24.1305 SI 21.9197 SD 9.5444 SU 7.6705 DU 4.0847 SPC	200.0118	5.8817	21.2101 SI 20.5649 SD 12.9313 SU 11.4035 DU 7.4034 RF
	Operate on parameters	53.1842	4	21.3261 DU 15.1905 SI 14.6461 SD 13.4587 SU 5.5418 RF	247.1504	6.5038	24.0751 SI 21.2345 SD 12.2987 SU 9.951 DU 6.6259 RF
Static init blocks		328.25	9.125	22.8104 SI 22.0107 SD 13.1379 SU 10.0533 DU 4.9124 RF	197.9509	5.7914	22.6027 SI 20.7587 SD 11.2843 SU 10.6552 DU 6.6975 RF

The only class that contains static non-final attributes, ToStringBuilder, has suffered 280 fine-grained source code changes over 33 commits. The number of modifications is higher than the average for similar classes (203.6) and there are more changes per commit (8.4848 vs. 5.9353). However, the top 5 change types are on par with those of other classes, though the percentages are very different.

The change-proneness of classes with constants is comparable to that of similar classes. Although there are, on average, more fine-grained changes (84.5472 vs. 72.7692), they were performed in fewer commits as the number of modifications per commit is higher (6.7925 vs. 4.5692). The top 5 change types are the same, even the order and the percentages are very similar.

The only singleton present in the latest version of Commons Lang, ObjectToStringComparator, did not suffer any fine-grained source code changes throughout the project's lifespan. Therefore, it is impossible to assess the impact of singletons on change- / defect-proneness for this system.

There have been, on average, 5 times more changes performed on utility classes (516.7907 vs. 98.9844) than on classes that are similar to them. This is due to a series of classes that are frequently modified, such as ArrayUtils (3224 changes), StringUtils (4866), or NumberUtils (2646). The number of changes per commit is also higher (9.1628 vs. 4.8672). Commons Lang is a system in which the discrepancy between the classes that contain other types of static methods is huge. The ones with methods that access state are more change-prone than classes which are similar to them in terms of size and complexity. The average number of modifications is almost triple (545 vs. 200.0118) while the number of changes per commit is double (11.8839 vs. 5.8817). On the other end of the spectrum are the classes with static methods that only operate on parameters. For this category the average number of changes is almost 5 times smaller (53.1842 vs. 2471504) than that of similar classes, but the number of modifications per commit is only a bit lower (4 vs. 6.5038). However, with regard to the top 5 change types, both cases are very similar. For the first category the top 4 changes are exactly the same while for the latter all 5 change types resemble the ones for similar classes (albeit in a different order).

Similar to utility classes, both the average number of changes (328.25 vs. 197.9509) and the number of changes per commit (9.125 vs. 5.7914) are higher for the classes that contain static initialization blocks compared to other classes. Finally, the top 5 change types are the same in terms of order and even percentages.

Table 5.4.3.2: Defect-proneness of classes with static constructs vs. similar classes for Commons Lang

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	141	14.1	32.6241 DU 19.1489 SD	85.5	7.8452	24.8677 SI 22.7096 SD

				14.1844 SI 11.3475 SU 4.2553 RF			10.22 DU 8.7162 SU 5.0404 SPC
	Constants	26.7755	9.8571	24.0985 SD 23.68 SI 10.0129 DU 8.7733 SU 5.1513 SPC	30.8611	5.2778	28.4428 SD 16.2016 SI 14.2214 DU 8.7309 SU 6.1206 APD
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	156.4483	9.8621	24.4655 SI 22.9888 SD 10.5356 DU 10.0507 SU 5.4882 CEC	49.75	6.9107	27.6382 SD 19.4185 SI 10.8399 DU 6.6762 SU 5.6712 SPC
	Access state	100.5	12.5625	29.6389 SD 20.8626 SI 8.4754 DU 6.0181 SU 5.3661 SPC	41.3768	7.9275	23.1751 SI 22.9311 SD 11.4656 DU 9.7955 SU 5.2543 CEC
	Operate on parameters	55.8571	6.4286	21.4834 SD 16.8798 DU 16.3683 SI 8.1841 SU 7.9284 SPC	88.8718	8.0513	24.9423 SD 22.8938 SI 10.3001 DU 8.7998 SU 4.8615 SPC
Static init blocks		91.75	13.875	34.7411 SD 23.5695 SI 5.7221 SU 5.3133 DU 5.0409 SPC	85.5714	7.2987	23.6455 SD 22.4313 SI 11.246 DU 9.1069 SU 5.0235 SPC

There are 10 bug-fix commits in which the only class with static non-final attributes, ToStringBuilder, was modified. In total, 141 fine-grained changes were performed on it, a value that is higher than the average for similar classes (85.5). The amount of changes per commit is also roughly double (14.1 vs. 7.8452), but there are more modifications related to documentation (almost 33% are doc updates) than for the other classes.

Just as for change-proneness, the defect-proneness of the classes that contain constants is not higher than that of similar classes. The average number of changes is comparable (26.7755 vs. 30.8611) and the number of modifications per commit is higher (9.8621 vs. 6.9107), thus indicating that they were altered in fewer bug-fix commits. The top 4 change types are the same and their percentages are also very similar.

There have been, on average, more modification performed on utility classes compared to similar classes (156.4483 vs. 49.75), but the ratio is smaller than for change-proneness. The difference between the number of changes per commit is also lower (9.8621 vs. 6.9107). The top 5 change types differ, especially in terms of percentages; for example, even though statement updates are fourth in both rankings, the percentages are quite different (10.0507 vs. 6.6762). Similarly to what was discovered for change-proneness, the classes with static methods that access state are much more error-prone than other similar classes, while the ones with static methods that solely operate on parameters are not. The former have, on average, more than double the number of changes (100.5 vs. 41.3768) and slightly more modifications were performed on them per commit (12.5625 vs. 7.9275). For the latter there are fewer modifications overall (55.8571 vs. 88.8718) and the amount of changes per commit is similar (6.4286 vs. 8.0513); this shows that they were altered in fewer commits. The top 4 types of changes are exactly the same in both cases and they resemble the ones for similar classes.

There is barely any difference between the average number of modifications for classes that contain static initialization blocks and other classes (91.75 vs. 85.5714). However, the number of changes per commit is higher for the first category (13.875 vs. 7.2987), therefore indicating that there are fewer bug-fix commits in which classes with initialization blocks were changed. The top 5 change types are the same, even the order is almost identical; there are however some differences percentage-wise.

5.4.4. Commons Math

Table 5.4.4.1: Change-proneness of classes with static constructs vs. similar classes for Commons Math

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence top 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence top 5 change types**
Static attributes	Non-final	129.3333	8	38.1443 SD 13.4021 SI 8.7629 CD 7.732 SU 5.9278 DU	94.7475	6.0777	18.4892 SI 18.4459 SD 15.7329 SU 11.4143 DU 6.4468 RF
	Constants	40.0198	8.332	20.2512 SI 19.9520 SD 16.5194 SU 11.3112 DU 5.1489 RF	65.0884	5.7569	16.4163 SD 15.856 SI 14.5197 SU 11.6162 DU 8.2209 RF
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	294.5	8.6154	20.7784 SD 20.0862 SI 12.8249 SU 11.114 RF 8.058 AF	86.0612	6.7211	18.216 SD 18.1804 SI 16.0893 SU 12.0307 DU 5.713 RF
	Access state	174.6667	7.8057	21.6671 SD 18.9324 SI 18.2225 SU 10.781 DU 5.4694 SPC	87.6378	6.5996	20.0087 SD 19.3268 SI 14.6409 SU 12.6963 DU 5.1956 RF
	Operate on parameters	67.7	7.6333	18.2568 SI 18.0282 DU 16.776 SD 13.8839 SU 6.0326 RF	82.6587	6.5326	21.982 SD 19.6539 SI 15.3065 SU 10.0118 DU 4.9391 RF
Static init blocks		534.6667	11.3333	23.1614 SD 19.8266 SI 15.1244 SU 9.1478 DU 6.6297 RF	90.57	6.743	19.5801 SD 19.2271 SI 14.9962 SD 13.2811 DU 5.1405 RF

The 3 classes that have static non-final attributes appear to be more change-prone compared to other similar classes. More fine-grained changes were performed on them on average (129.3333 vs. 94.7475) and the number of modifications per commit is higher (8 vs. 6.0777). The top 5 change types are also different; for example, over 38% of the changes are statement deletes while for similar classes this percentage is around 18.5%.

The classes that contain constants suffered less changes compared to other classes. The average number of modifications is lower (40.0198 vs. 65.0884) while the number of changes per commits is higher (8.332 vs. 5.7569), thus indicating that they were altered in significantly fewer commits. The top 5 change types resemble the ones observed for other production classes.

No fine-grained source code changes were encountered for Decimal64Field, the only singleton found in the last version of Commons Math. Because of this, we are not able to determine its effect on change- / defect-proneness.

Utility classes were changed more frequently compared to other classes that are similar to them in terms of size and complexity (average number of changes of 294.5 vs. 86.0612). This is mainly because certain instances are central to the system and have suffered numerous modifications throughout its lifetime; examples include ComplexUtils (1103 changes), MathUtils (1223), and FastMath (2329). However, the number of changes per commit is not so different (8.6154 vs. 6.7211), thereby suggesting that the utility classes were modified in many more commits. Similar to what was observed for the previous project, the classes with static methods that access state are more change-prone than other classes while the ones with static methods the only operate on parameters are not. For the first category the average number of changes is significantly higher (174.6667 vs. 87.6378) while the amount of modifications per commit is comparable (7.8057 vs. 6.5996). On the other hand, for classes with static methods that solely operate on parameters the averages are completely different (67.7 vs. 82.6587) and the number of changes per commit are similar (7.6333 vs. 6.5326). In both cases the top 4 change types are the same as for similar classes; for the latter even the fifth one, REMOVED_FUNCTIONALITY, concurs.

The classes that contain at least 1 static initialization block are also changed more compared to similar classes (an average of 534.6667 vs. 90.57 changes). The results might be skewed because there are only 6 instances and FastMath (with 2329 changes) is one of them. Furthermore, the average number of modifications per commit is also higher (11.3333 vs. 6.743).

Table 5.4.4.2: Defect-proneness of classes with static constructs vs. similar classes for Commons Math

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	55.6667	6	14.2857 DAC 10.3896 DU 10.3896 SD 9.0909 CD 7.7922 AF	46.3631	7.5706	18.144 SD 17.2862 SI 15.1666 SU 13.5194 DU 5.7869 RF
	Constants	29.8772	8.5848	22.0139 SI 17.8826 SD 16.3297 SU 14.6303 DU 4.2289 SPC	33.9171	6.6188	16.5011 SD 13.1292 SU 11.5654 SI 11.4514 DU 8.9102 AF
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	101.5789	9.1053	27.3057 SD 17.3057 SI 12.1762 RF 10.1036 SU 5.544 SPC	42.9006	7.4699	18.1844 SI 15.8815 SD 15.7972 SU 14.6669 DU 5.343 AF
	Access state	43.2	4.8	20.4216 SI 19.3676 SD 15.0198 SU 11.1989 DU 9.8814 AF	42.1617	6.8271	20.5617 SD 19.465 SI 16.9594 DU 12.4476 SU 4.8239 SPC
	Operate on parameters	48.4792	7.3333	36.055 DU 15.6854 SI 12.1186 SD 11.0443 SU	42.1026	6.6624	22.5944 SD 20.4324 SI 13.0532 SU 11.7946 DU

				5.0279 RF			5.5319 SPC
Static init blocks		214.8	15	31.9367 SD 18.2495 SI 8.2886 CD 8.1937 SU 8.0074 SPC	43.742	7.4493	18.077 SI 16.2083 SD 15.6186 SU 14.2867 DU 5.9042 RF

All 3 classes that contain static non-final attributes were modified during bug-fix commits and the average number of changes is higher than for classes which are similar to them in terms of size and complexity (55.6667 vs. 46.3631). However, the number of changes per commit is lower (6 vs. 7.5706), thus indicating that there are more commits in which they were fixed. The top 5 change types are very different compared to what was observed thus far; it is one of the few cases in which statement insertions and updates do not appear in this ranking.

There is no significant difference between the average number of changes for classes with constants and other similar classes (29.8772 vs. 33.9171). However, the number of modifications per commit is higher for the former (8.5848 vs. 6.6188), thereby suggesting that they were changed less frequently. Four of the top 5 change types are the same, albeit the order and the percentages are slightly different.

Nineteen utility classes were modified during bug-fix commits. Even though the average number of changes for utility classes is higher than for similar classes (101.5789 vs. 42.9006), the difference is smaller compared to what was observed for change-proneness. Additionally, comparable values were obtained in terms of number of changes per commit (9.1053 vs. 7.4699). However, the top 5 change types are different both in order and percentage-wise. The classes with static methods that access state are a bit more defect-prone compared to similar classes. The average number of modifications is higher (43.2 vs. 42.1617) while the amount of changes per commit is lower (4.8 vs. 6.8271), thus suggesting that they were modified in more bug-fix commits. For the classes with static methods that only operate on parameters, even though the average is slightly higher (48.4792 vs. 42.1026), the number of changes per commit is also higher (7.3333 vs. 6.6624); this indicates that the number of bug-fix commits in which they were changed is roughly the same. Similar to change-proneness, the top 4 change types are the same, although the order and the percentages are quite different (especially for the latter category).

The observations that can be made with regard to the error-proneness of the classes that contain static initialization blocks are similar to the ones for utility classes. Although the difference between the average number of changes is very high (214.8 vs. 43.742), it is much lower than for change-proneness. The number of modifications per commit is double for this kind of classes compared to similar classes (15 vs. 7.4493). This is mainly due to 1 class, FashMath, which suffered 929 changes over 28 bug-fix commits. The percentages for the top 5 change types are also very different; for example, almost 32% of the changes are statement deletes for classes with static initialization blocks while for the other classes the percentage is just over 16%.

5.4.5. Digester

Table 5.4.5.1: Change-proneness of classes with static constructs vs. similar classes for Digester

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	-	-	-	-	-	-
	Constants	66.75	3.375	19.5521 SD 16.8389 SU 14.6856 DU 13.4798 SI	61.3654	3.7404	21.1376 SU 18.5522 SI 15.8258 SD 10.0125 DU

				7.5797 RF			5.0611 CEC
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	6	2.625	41.6667 SU 16.6667 ROS 16.6667 RF 16.6667 IAC 8.3333 DU	68.9841	3.6905	19.9609 SU 17.2227 SI 16.8431 SD 11.2632 DU 5.6143 RF
	Access state	-	-	-	-	-	-
	Operate on parameters	103.1667	4.3333	22.1325 SI 20.0323 SD 15.5089 SU 8.7237 DU 8.5622 RF	66.2705	3.6393	20.334 SU 16.8213 SI 16.5739 SD 12.4533 DU 5.4051 RF
Static init blocks		-	-	-	-	-	-

There are no singletons or production classes with static non-final attributes / initialization blocks in the latest version of Digester. The change-proneness of the classes that contain constants is comparable to that of other classes; both the average number of changes (66.75 vs. 61.3654) and the number of modifications per commit (3.375 vs. 3.7404) are very similar. Four of the top 5 change types are the same, although the percentages are quite different.

The 2 utility classes, AnnotationUtils and LogUtils, suffered only a small number of changes (9 and 3, respectively) compared to other classes. The amount of modifications per commits is also lower (2.625 vs. 3.6905). In terms of change types, they are very different; two of the most common change types, statement insertions and deletions, were not performed on the aforementioned classes. There are no classes with static methods that access their state. Those with static methods that operate only on parameters are slightly more change-prone than classes that are similar to them in terms of size and complexity. They have, on average, a higher number of changes (103.1667 vs. 66.2705), but there are also more changes per commit (4.3333 vs. 3.6393). The top 5 modification types are the same as for similar classes, albeit the order and percentages do differ.

Table 5.4.5.2: Defect-proneness of classes with static constructs vs. similar classes for Digester

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	-	-	-	-	-	-
	Constants	12.3125	3.125	25.5539 SD 21.5657 DU 15.0665 SI 11.226 SU 4.579 RF	18.0141	3.8451	27.5997 SI 21.1102 SD 18.1392 DU 10.4769 SU 3.8311 SPC
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	-	-	-	-	-	-
	Access state	-	-	-	-	-	-
	Operate on parameters	22.2	2.8	24.3243 DU 20.7207 SI 19.8198 SD 9.9099 SU 7.2072 RF	22.5	3.7683	23.4146 SI 22.8184 SD 19.0244 DU 10.7859 SU 4.0108 SPC
Static init blocks		-	-	-	-	-	-

There are no static non-final attributes in the production code. The classes with constants are not more error-prone compared to other similar classes; they were changed less in bug-fix commits (average number of modifications of 12.3125 vs. 18.0141) and the number of changes per commit are close (3.125 vs. 3.8451). The top 4 change types are the same, but their order and percentages are different.

The changes performed on the 2 utility classes, AnnotationUtils and LogUtils, did not occur during bug-fix commits. No static methods that access state were found. The classes with static methods that only operate on parameters are not more error-prone that other similar classes. The average number of modifications is almost identical (22.2 vs. 22.5) and there are a comparable number of changes per commit (2.8 vs. 3.7683). The top 4 change types are also the same, the only differences are in terms of order and percentages.

5.4.6. Geode

Table 5.4.6.1: Change-proneness of classes with static constructs vs. similar classes for Geode

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	847.1574	9.3333	20.383 SI 19.7873 SD 13.3518 SU 8.1689 SPC 5.0113 CEC	252.2445	7.2836	16.1517 SD 15.843 SI 13.5458 SU 5.098 DU 4.7261 CEC
	Constants	112.4557	9.252	18.6819 SI 18.2977 SD 14.7422 SU 6.0478 SPC 5.7684 CEC	148.7544	5.5167	11.7809 IAC 11.4253 SI 11.1711 SD 10.1268 SU 9.0795 DAC
Singletons	Stateful	431.2667	6.9333	30.3756 SU 14.5308 SI 11.1145 SD 9.0122 DU 5.3022 CEC	279.5486	7.3865	17.78 SD 17.4616 SI 12.3458 SU 5.1266 SPC 4.7614 CEC
	Stateless	496.4808	3.6731	24.4606 SU 16.1599 SPC 12.1625 SI 11.9417 SD 10.6209 SOC	275.5052	7.4701	15.9505 SI 15.5954 SD 15.0576 SU 4.8966 DU 4.8827 CEC
Static methods	Utility classes	203.4145	4.6891	16.2867 SI 16.0626 SD 13.3014 ROS 12.9448 AOS 7.9116 SPC	287.5462	7.6296	16.788 SI 16.43 SD 14.0658 SU 4.9194 SPC 4.8774 DU
	Access state	964.7198	10.9066	21.1255 SI 20.728 SD 12.6103 SU 9.3616 CEC 6.3419 SPC	221.1674	7.0777	15.1035 SI 14.7718 SD 13.8617 SU 5.7841 DU 4.8912 RF
	Operate on parameters	413.2644	5.5076	20.0649 SU 17.5532 SI 17.2656 SD 6.3178 SPC 5.6353 PTC	258.1554	7.7	16.5422 SI 16.1757 SD 11.7498 SU 5.2278 DU 4.9318 RF
Static init blocks		1113.6667	15.7576	17.5016 SI	255.7004	7.1338	17.6605 SI

			17.4145 SD 12.1643 CEC 10.5616 SU 6.7236 SPC			16.2767 SD 12.9021 SU 5.1585 DU 4.8927 SPC

Classes with static non-final attributes appear to be more change-prone than other similar classes, while the ones that contain only constants are not. For the first category, the average number of changes is almost 3.5 time higher (847.1574 vs. 252.2445) while the amount of modifications per commit are comparable (9.3333 vs. 7.2836). On the other hand, the average for classes with constants is lower (112.4557 vs. 148.7544) and the number of changes per commit is almost double (9.252 vs. 5.5167); this indicates that they were modified in fewer commits. The top 5 change types are the same for both categories; only the statement-level ones appear in the rankings for similar classes. An interesting observation can be made with regard to the top modification for the classes that are similar in terms of size and complexity to the ones with constants; its type is INCREASE_ACCESSIBILITY_CHANGE and represents 11.78% of the total number of modifications that were performed.

Both types of singletons have a much higher change-proneness than similar classes. The average number of changes is greater (431.2667 vs. 279.5486 and 496.4808 vs. 275.5052, respectively) and the amount of changes per commit is lower (3.6731 vs. 7.4701 and 4.6891 vs 7.6296) compared to the other classes; this implies that they were modified in many more commits. With regard to the top 5 change types, only the 3 most common ones (statement insertions, deletions, and updates) are present in all the rankings.

The change-proneness of utility classes is comparable to that of other classes. Although less modifications were performed on them on average (203.4145 vs. 287.5462), the number of changes per commit is also lower (4.6891 vs. 7.6296); thus the amount of commits in which they were changed is more or less the same. In terms of top 5 change types, this is one of the few cases in which additional / removed object state modifications appear in the list (ranked third and fourth); because of this, only the first 2 change types resemble the ones for similar classes.

The rest of the classes that contain static methods are also more change-prone than other similar classes, especially the ones with static methods that access state. These classes have, on average, almost 4.5 times more changes (964.7198 vs. 221.1674) while the amount of modifications per commit is comparable to that of similar classes (10.9066 vs. 7.0777). The classes with static methods that operate on parameters are also a bit more change-prone. When compared to other classes, more changes were performed on them (413.2644 vs. 258.1554) and the number of modification per commit is lower (5.5076 vs. 7.7); this shows that they were modified in more commits. In term of top 5 change types, the first 3 are the same both for the 2 categories of classes with static methods and for other classes that are similar to the instances in terms of size and complexity.

Finally, the classes with static initialization blocks suffered roughly 4 time more changes (1113.6667 vs. 255.7004) than similar classes and there are twice as many modifications per commit (15.7576 vs. 7.1338); this indicates that their change-proneness is higher. Four of the top 5 change types are the same, even the percentages are very similar.

Table 5.4.6.2: Defect-proneness of classes with static constructs vs. similar classes for Geode

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	560.6947	9.8947	20.4408 SI 19.932 SD 12.462 SU 8.7692 SPC 5.266 CEC	153.3238	9.6025	15.7888 SD 15.6849 SI 15.1362 SU 5.2902 DU 5.0906 AF
	Constants	130.416	13.2471	17.9384 SD	100.0837	4.6946	13.6287 SI

				17.4154 SI 15.8108 SU 6.3485 SPC 5.8649 CEC			11.9873 SD 11.1341 SU 9.8444 DU 8.0119 AF
Singletons	Stateful	247.0769	7.4615	29.67 SU 15.0685 SI 10.2117 SD 9.5268 DU 5.1059 CEC	174.6582	9.6389	17.5633 SD 17.5136 SI 13.5224 SU 6.5048 SPC 5.0831 AF
	Stateless	276.551	3.6939	23.9244 SU 16.8401 SPC 12.191 SI 11.1505 SD 9.6229 SOC	172.3137	9.7911	15.7411 SI 15.6948 SD 15.2579 SU 6.2717 AF 5.1622 CEC
Static methods	Utility classes	171.1538	5.3615	16.2966 ROS 16.0315 SI 14.5888 SD 12.809 AOS 7.9506 SPC	175.4005	9.9539	16.6446 SD 16.5346 SI 15.5172 SU 5.3145 AF 5.2864 SPC
	Access state	495.6746	9.8462	22.2063 SD 16.9621 SI 13.5492 SU 9.2648 CEC 7.2986 SPC	141.505	9.5995	16.3281 SI 15.0945 SU 14.3962 SD 5.7665 DU 5.2545 AF
	Operate on parameters	231.519	5.1044	19.8619 SI 18.8518 SU 18.0071 SD 6.2876 SPC 5.801 PTC	163.0041	10.6003	16.0341 SD 15.4655 SI 13.3967 SU 5.8077 AF 5.5012 DU
Static init blocks		402.2453	14.8302	19.4148 SI 14.807 SD 10.6479 CEC 10.1427 SU 8.6918 SPC	158.9849	9.4629	16.7274 SD 16.1027 SI 15.2946 SU 5.2242 DU 5.0403 SPC

The classes with static non-final attributes are more error-prone than other similar classes, while for the ones that contain constants defect-proneness is lower. For the former the average number of modifications is more than 3.5 times higher (560.6947 vs. 153.3238) and the amount of changes per commit is the same (9.8947 vs. 9.6025). For the second category, although there are slightly more changes (130.416 vs. 100.0837), the number of modifications per commit is roughly 3 times higher (13.2471 vs. 4.6946); therefore they were altered in fewer bug-fix commits. The top 5 modification types are the same in both cases, but only the first 3 (the statement-level ones) also appear in the rankings for similar classes.

Similar to what was observed for change-proneness, the singletons are also more error-prone. Both the stateful and the stateless ones have, on average, a higher number of changes (247.0769 vs. 174.6582 and 276.551 vs. 172.3137, respectively) than other similar classes and the amount of modifications per commit is lower (7.4615 vs. 9.6389 and 3.6939 vs. 9.7911), thus proving that they were changed in fewer bug-fix commits. Three of the top 5 change types are the same, the statement-level ones, but even for them the percentages are higher for the singletons.

The defect-proneness of utility classes is also similar to that of other classes. The average number of changes is almost the same (171.1538 vs. 175.4005), but the number of modifications per commit is smaller (5.3615 vs. 9.9539). The top 5 change types in bug-fix commits differ as well due to the additional / removed object state modifications that were performed on utility classes.

Just as for change-proneness, the other classes with static methods are more defect-prone than similar classes. For the ones with methods that access their state, the average number of modifications is more than 3.5 times higher (495.6746 vs. 141.505) while the number of changes per commit is almost the same (9.8462 vs. 9.5995). For the other category, classes with static methods that only operate on parameters, the error-proneness is not significantly higher than that of similar classes; there were, on average, more changes performed on them (231.519 vs. 163.0041) and the number of modifications per commit is lower (5.1044 vs. 10.6003). Finally, while the top 3 modification types are the same (statement-level changes), their order and percentages are different.

Although classes that contain static initialization blocks are more defect-prone compared to similar classes, their error-proneness is lower than their change-proneness. They suffered, on average, 2.5 times more modifications (402.14.8302 vs. 158.9849), but more changes were performed on them per commit (14.8302 vs. 9.4629). In terms of top 5 change types, 4 of them are the same, albeit the order and percentages are significantly different.

5.4.7. jHotDraw

Table 5.4.7.1: Change-proneness of classes with static constructs vs. similar classes for jHotDraw

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	91.75	11.3445	23.4332 SD 14.9864 SI 11.7166 SU 10.3542 RF 5.4496 AF	88.8229	6.92	21.288 SD 16.4143 SU 16.1698 SI 7.5775 RF 4.6999 SPC
	Constants	104.8304	6.5536	19.7513 SD 16.387 SI 15.3224 SU 8.3042 RF 5.5106 SPC	62.2388	7.791	25.8034 SD 15.3477 SU 11.7266 SI 9.4964 RF 7.0983 DU
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	-	-	-	-	-	-
	Access state	78	8.6667	21.7949 AF 20.5128 SI 12.8205 RF 10.2564 SU 8.9744 SD	88.9494	7.0112	21.3983 SD 15.354 SU 15.1393 SI 8.596 RF 5.6528 SPC
	Operate on parameters	34.1667	5.4306	24.878 SD 18.5366 SI 12.1951 SU 11.2195 DU 10.7317 RF	90.7861	7.0867	22.2912 SD 14.3699 SU 14.1216 SI 8.5891 RF 6.6221 SPC
Static init blocks		539	10.7105	26.6234 SD 20.0371 SI 18.2746 SU 7.9777 SPC 3.8961 CEC	83.8023	6.9831	20.9533 SD 15.1149 SU 14.8116 SI 9.0137 RF 5.8181 DU

Surprisingly, the classes that contain static non-final attributes are not more change-prone than other similar classes. The average number of modifications is a bit higher (91.75 vs. 88.8229), but so is the amount of changes per commit (11.3445 vs. 6.92). Unlike for the previous systems, for jHotDraw the classes with constants have a higher change-proneness than other similar classes. Their average number of changes is greater (104.8304 vs. 62.2388) and there are fewer modifications per commit (6.5536 vs. 7.791).

No stateful singletons were found in the latest version of the project. Additionally, the only stateless singleton (FigureLayerComparator) did not suffer any fine-grained changes throughout its existence. Neither did any of the 4 utility classes. Only 1 class that contains static methods that access state was altered (AbstractDrawing with 78 changes in 9 commits); this class is a bit less change-prone compared to the classes that were categorized as similar to it. The 6 classes with static methods that solely operate on parameters have a much lower change-proneness. Their average number of modifications is almost 3 times smaller (34.1667 vs. 90.7861) while the number of changes per commit is comparable (5.4306 vs. 7.0867).

Two classes with static initialization blocks were modified, DefaultDrawingView (949 changes in 61 commits) and AttributeKeys (129 changes in 22 commits). It can be observed that the number of modifications is very high, thus showing that they are more change-prone than the other classes.

Table 5.4.7.2: Defect-proneness of classes with static constructs vs. similar classes for jHotDraw

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	29	22.5	32.7586 SI 29.3103 SD 17.2414 SU 3.4483 PD 3.4483 SPC	19.9195	9.4253	26.5626 SD 17.1766 SI 15.0415 SU 7.7513 RF 6.232 SPC
	Constants	24.0545	8.6727	25.0945 SD 17.7627 SI 15.5707 SU 6.576 RF 6.5004 SPC	13.7647	11.4118	27.3504 SD 21.1538 SI 17.5214 SU 6.8376 RF 5.7692 DU
Singletons	Stateful	-	-	-	-	-	-
	Stateless	-	-	-	-	-	-
Static methods	Utility classes	-	-	-	-	-	-
	Access state	26	13	53.8462 SI 11.5385 SD 11.5385 SU 7.6923 PD 7.6923 RF	20.0568	9.6818	25.8924 SD 18.1303 SI 16.1473 SU 6.6289 RF 6.2323 SPC
	Operate on parameters	6	6	66.6667 SI 16.6667 SD 16.6667 SU	20.2841	9.7614	25.7143 SD 18.4874 SI 16.0784 SU 6.6667 RF 6.1625 SPC
Static init blocks		106.5	12.8555	27.6995 SD 23.9437 SU 10.7981 SI 9.3897 SPC 3.7559 DU	18.1379	9.6552	25.4119 SD 19.7085 SI 15.019 SU 7.2243 RF 5.7034 SPC

Only 2 classes that contain static non-final attributes were changed during bug-fixing, AbstractDrawing (26 changes in 2 commits) and ColorIcon (32 changes in 1 commit). Though the average number of changes is higher (29 vs. 19.9195), so is the amount of modifications per commit (22.5 vs. 9.4253); this indicates that they were altered in a comparable number of bug-fix commits. Same as for change-proneness, classes with constants are also more error-prone. They have, on average, a higher number of changes (24.0545 vs. 13.7647) and the amount of changes per commit is lower (8.6727 vs. 11.4118).

The only class with static methods that access state suffered 26 fine-grained changes over 2 bug-fix commits; these values are comparable to the ones obtained for the classes that were considered similar to it. RelativeLocator is the only class with static methods that operate on parameters that was changed during bug-fixing. Only 6 modifications were performed on it in a single commit, much fewer than for similar classes.

Unsurprisingly, the 2 classes that contain static initialization blocks are also more error-prone. The average number of changes is almost 6 times higher (106.5 vs. 18.1379) while the amount of modifications per commit is slightly greater (12.8555 vs. 9.6552).

5.4.8. Pig

Table 5.4.8.1: Change-proneness of classes with static constructs vs. similar classes for Pig

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	79.8333	9.6667	30.5047 SD 21.5277 SI 9.5911 SU 6.9016 SPC 4.728 RF	40.975	6.3524	27.3827 SD 18.8605 SI 8.845 SU 7.8542 SPC 5.0469 RF
	Constants	44.4253	7.6851	28.84 SD 19.0464 SI 10.3782 SU 7.5034 SPC 5.3353 SOC	23.017	6.1473	26.4 SD 21.00092 SI 8.0862 SU 7.8892 SPC 6.7446 RF
Singletons	Stateful	68.3	7.1	23.8653 SD 13.7628 SI 12.0059 SU 8.0527 SPC 8.0527 SOC	46.7092	6.8646	29.3028 SD 20.693 SI 8.7296 SU 6.5953 SPC 4.9735 RF
	Stateless	4.3333	3.3333	30.7692 SD 23.0769 RF 15.3846 AF 15.3846 SI 7.6923 PID	47.2314	6.8843	28.2041 SD 20.5643 SI 9.7838 SU 6.6085 SPC 4.9467 RF
Static methods	Utility classes	64.26	8.4	29.225 SD 22.9381 SI 9.2126 SU 6.4737 SPC 6.3803 RF	55.5565	6.7381	28.0833 SD 19.1701 SI 9.8437 SU 7.7349 SPC 4.7997 RF
	Access state	81.8431	7.1961	34.6191 SD 17.5371 SI 9.2717 SU 6.9957 RF 5.2228 SPC	44.1215	6.8407	27.2088 SD 19.8772 SI 10.8586 SU 6.9754 SPC 4.6371 RF
	Operate on parameters	62.2911	6.9494	29.2278 SD 19.5995 SI 11.3153 SU 8.9151 SPC 4.2655 RF	40.883	6.8571	26.8912 SD 18.5512 SI 10.3083 SU 9.2033 SPC 5.4688 RF
Static init blocks		98	8.619	27.794 SD 22.3518 SI	45.2969	6.8063	28.2304 SD 19.3618 SI

			11.2245 SU 9.8154 SPC 6.3168 RF			9.6757 SU 7.4474 SPC 5.1052 RF

The average number of changes for classes with static non-final attributes is almost double than for other similar classes (79.8333 vs. 40.975). The number of changes per commit is also higher (9.6667 vs. 6.3524) and the top 5 change types are identical (even the percentages are very close).

Surprisingly, similar observations can be made for classes that contain constants. The average number of modifications is once again double (44.4253 vs. 23.017), but the number of changes per commit is closer to the one obtained for other classes (7.6851 vs. 6.1473). The top 4 change types are the same and their percentages differ by a small margin.

Ten of the 14 stateful singletons were altered throughout the history of Pig. They appear to be more change-prone compared to similar classes; the average number of changes is higher (68.3 vs. 46.7092), while the amount of modifications per commit is roughly the same (7.1 vs. 6.8646). Similar to before, the top 4 change types are identical, albeit the percentages are significantly different. Three of the 4 stateless singleton have also suffered fine-grained changes. However, their corresponding measurements are much lower than for similar classes (4.3333 vs. 47.2314 and 3.3333 vs 6.8843, respectively) or their stateful counterparts. The top 5 change types are also very different than what was observed for the other static constructs; for example, it is the first time PARENT_INTERFACE_DELETE appeared in the rankings.

The change-proneness for utility classes is comparable to that of similar classes. Though their average number of changes is higher (64.26 vs. 55.5565), the amount of modifications per commit is also greater (8.4 vs. 6.7381); this indicates that they were altered in roughly the same number of commits. The top 5 change types are also identical, even the percentages are almost the same.

The other classes that contain static methods are more change-prone compared to similar classes. Both the ones that have methods that access their state and those with static methods that only operate on parameters have suffered, on average, a higher number changes (81.8431 vs. 44.1215 and 62.2911 vs. 40.883, respectively) and the number of modifications per commit are almost identical (7.1961 vs. 6.8407 and 6.9494 vs. 6.8571). Furthermore, the top 5 change types are also the same (even in terms of order); there are only small differences with regard to percentages.

Classes with static initialization blocks seem more change-prone that other which are similar to them. The average number of modifications is more than double (98 vs. 45.2969) and the number of changes per commit is only a bit higher (8.619 vs. 6.8063). Finally, the top 5 change types are again the same; this situation was encountered several time for different categories of static constructs.

Table 5.4.8.2: Defect-proneness of classes with static constructs vs. similar classes for Pig

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	37.1744	8.6163	36.0963 SD 14.0131 SI 10.7914 SU 8.508 SPC 4.8796 SOC	22.3587	5.6675	28.535 SD 18.4851 SI 10.0712 SU 9.7525 SPC 5.7049 CEC
	Constants	25.1836	6.3281	29.3105 SD 17.9305 SI 10.6251 SU 9.3483 SPC 5.8732 SOC	14.3546	6.004	33.3056 SD 15.9034 SI 9.6586 SPC 9.3256 SU 7.6603 CEC
Singletons	Stateful	34.3333	8.4444	23.6246 SD 11.9741 SPC 11.6505 SU	24.7008	6.1265	31.6235 SD 16.5108 SI 11.2187 SU

				11.0032 SI 11.0032 SOC			8.3732 SPC 5.3085 CEC
	Stateless	3.5	3.5	57.1429 SD 28.5714 SI 14.2857 APD	24.9564	6.1782	30.4372 SD 17.3451 SI 11.2595 SU 9.4422 SPC 4.2607 CEC
Static methods	Utility classes	33.4651	6.7442	33.148 SD 18.1376 SI 11.1188 SU 8.4781 SPC 4.934 RF	24.0754	6.1142	30.1047 SD 17.25 SI 10.1423 SU 9.5605 SPC 5.2994 CEC
	Access state	37.5	6.5	36.8696 SD 12.9275 SI 10.7826 SU 6.6667 SOC 6.4928 SPC	23.6117	6.1345	29.435 SD 18.0524 SI 10.17 SU 9.9035 SPC 5.5214 CEC
	Operate on parameters	14.4444	5.2698	27.8214 SD 18.1786 SI 11.25 SPC 11 SU 5.4286 CEC	22.0946	6.295	31.2029 SD 17.1152 SI 10.0408 SU 8.9195 SPC 5.2803 SOC
Static init blocks		41.3684	6.2632	32.4427 SD 14.7583 SPC 14.2494 SU 14.1221 SI 5.598 CEC	24.2295	6.1639	30.3197 SD 16.566 SI 9.9882 SU 9.0832 SPC 6.2351 CEC

Just as for change-proneness, the average number of modifications in bug-fix commits is higher for classes that contain static non-final attributes than for similar classes (37.1744 vs. 22.3587). The number of changes per commit is also higher (8.6163 vs. 5.6675). Finally, the top 4 change types are the same and the percentages are not very different either.

Classes with constants also seem a bit more defect-prone compared to the other classes. Both the average number of changes (25.1836 vs. 14.3546) and the number of modifications per commit (6.3281 vs. 6.004) are higher. Four of the top 5 change types are the same, all of them being statement-level changes, but the order and percentages are different.

With regard to singletons, 9 stateful and 2 stateless ones (SparkSims and DownloadResolver) were changed during bug-fixing activities. While the first appear to be a bit more error-prone compared to similar classes, those from the second category are not. The average number of modification and the number of changes per commit are higher for the former (34.3333 vs. 24.7008 and 8.4444 vs. 6.1265, respectively). For the stateless variants the corresponding measurements are very low (3.5 for both values). It is also worth noting that only 3 types of changes have been encountered in the bug-fix commits for stateless singletons (out of a total of 7 changes).

The defect-proneness for utility classes is a bit higher than for similar classes. The average number of changes is greater (33.4651 vs. 24.0754), while the amount of modifications per commit are close (6.7442 vs. 6.1142). The top 4 change types are identical, even the percentages are more or less the same.

The classes that contain static methods that access state are also more defect-prone that other similar classes. The average number of modifications is higher (37.5 vs. 23.6117) while the number of changes per commit is very close (6.5 vs. 6.1345). On the other hand, even though they were proven to be change-prone, classes with static methods that only operate on parameters are not more error-prone when compared to similar classes. They suffered, on average, a smaller number of changes (14.4444 vs. 22.0946) and the number of modifications per commit is also a bit lower (5.2698 vs. 6.295).

The classes with static initialization blocks have a defect-proneness similar to that of utility classes. When compared to other production classes, they were modified more (average number of changes of 41.3684 vs. 24.2295) and the same amount of changes were performed on them per bug-fix commit (6.2632 vs. 6.1639). The top 5 modification types are the same, but the order and the percentages are significantly different.

5.4.9. Spring Core

Table 5.4.9.1: Change-proneness of classes with static constructs vs. similar classes for Spring Core

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	713.5	8	23.8963 SI 23.0904 SD 14.7162 SU 11.0021 MR 7.288 ROS	844.4419	8.1938	24.339 SD 24.3234 SI 16.2217 SU 4.9283 DU 4.0488 CD
	Constants	154.9277	12.9518	29.6425 SD 29.6295 SI 14.3143 SU 4.3931 CD 4.3245 CI	419.3352	5.9832	16.9635 SU 16.9129 SI 16.9023 SD 10.7339 DU 6.5693 CEC
Singletons	Stateful	-	-	-	-	-	-
	Stateless	5070	809.9231	37.86 SI 37.7515 SD 5.9862 API 5.9467 APD 3.9645 CI	12.5	8.1577	23.7108 SI 23.7004 SD 16.9455 SU 6.0812 DU 3.9999 CD
Static methods	Utility classes	511.9348	7.7174	38.2946 SD 38.2691 SI 9.9537 SU 3.0362 APD 3.015 API	912.8287	8.2917	23.7591 SD 23.7565 SI 15.8436 SU 5.2569 DU 4.4403 CD
	Access state	5355.75	29.6984	26.5195 SD 25.7715 SI 11.1926 SU 4.0533 RF 3.9685 AF	676.5976	6.7967	25.1528 SI 24.9154 SD 16.527 SU 5.3514 DU 4.4814 CEC
	Operate on parameters	315.3158	6.7368	28.6763 SU 25.0876 SI 24.6703 SD 4.4734 SPC 4.2564 SOC	883.6584	8.3045	26.3277 SD 26.311 SI 14.8396 SU 5.9602 DU 5.0083 CD
Static init blocks		28.7647	6.3529	24.1309 SI 16.7689 SD 9.816 CEC 9.6115 RF 8.998 SU	898.902	8.3184	25.3289 SD 25.3075 SI 15.229 SU 4.8658 DU 4.0031 CD

From the 4 classes that have static non-final attributes 1 stands out, ResourceDecoder, which suffered 2663 fine-grained source code changes during 411 commits. These classes have, on average, a smaller number of modifications (713.5 vs. 844.4419) while the number of changes per commit is almost identical to that of similar classes (8 vs. 8.1938); this suggests that their change-proneness is lower. The situation is even clearer for production classes that have constants. For them the average is almost 3 times lower (154.9277 vs. 419.3352) and the number of modifications per commit is twice as high (12.9518 vs. 5.9832); thus they were altered in much fewer commits. Only the first 3 of top 5 change types are similar, the last 2 are completely different.

There are no stateful singletons in the latest version of Spring Core that was studied. Only 2 of the 5 stateless ones have undergone fine-grained changes and the results obtained for them are at opposite ends of the spectrum. AnnotationAwareOrderComparator has suffered a record of 10139 modifications over 411 commits, while ComparableComparator was changed only once. Because of this it is impossible to make a proper assessment with regard to the change-proneness of stateless singletons for this system.

The 46 utility classes have a lower change-proneness than classes which are similar to them in terms of size and complexity. The average number of changes is smaller (511.9348 vs. 912.8287) and the amount of modifications per commit is roughly the same (7.7174 vs. 8.2917). The top 3 change types are identical, but the following 2 (alternative part delete / insert) are not. Regarding the rest of the classes that contain static methods, 4 with methods that access state and 31 with static methods that solely operate on parameters have suffered fine-grained changes throughout Spring Core's lifetime. The most noticeable one in terms of number of modifications is Frame (18219 changes over 435 commits). It is from the first category, which causes the average number of changes for these instances (5355.75) to be much higher than for similar classes (676.5976) or for classes with static methods that only operate on parameters (315.3158). This observation also holds true for the number of modifications per commit (29.6984 vs. 6.7967 and 6.7368, respectively). With regard to the top 5 change types, only the statement-level ones appear in all 3 rankings.

The 17 classes that contain static initialization blocks have a much lower change-proneness than other similar classes. The average number of modifications is very low (28.7647 vs. 898.902) and the amount of changes per commit is comparable to that of similar classes (6.3529 vs. 8.3184). From the top 5 change types 3 appear in both lists, but the order and percentages are significantly different.

Table 5.4.9.2: Defect-proneness of classes with static constructs vs. similar classes for Spring Core

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	286.5	14.5	47.1663 SD 15.1129 SI 13.4189 SU 7.0637 CD 2.8953 APD	80.9423	5.75	25.7781 SD 16.6073 SU 10.5726 SI 8.9808 DU 6.795 CD
	Constants	72.75	12	36.0825 SI 15.4639 SD 10.6529 SU 9.2784 AOS 7.9038 MR	166.561	9.061	41.2505 SD 15.4604 SU 12.2669 SI 7.1167 CD 3.756 DU
Singletons	Stateful	-	-	-	-	-	-
	Stateless	755	25.1667	56.6887 SD 18.8079 SI 10.3311 APD 6.8874 CD 3.4437 AF	155.2235	9.0118	39.7984 SD 15.2039 SU 13.4531 SI 6.988 DU 3.8881 CD
Static methods	Utility classes	122	9.7333	50.5464 SD 26.776 SI 8.6339 SU 5.082 APD 2.7869 SPC	170.6901	9.0845	38.2277 SD 14.2388 SU 11.7749 SI 8.004 CD 4.101 DU
	Access	699.5	23.3734	42.2059 SD	138.8026	8.7763	40.2313 SD

	state			11.1177 SI 10.6471 SU 6.9706 AF 5.4412 CD			16.4327 SU 14.7407 SI 6.4794 CD 3.7918 DU
	Operate on parameters	69.5	9.25	35.9712 SU 27.3381 SI 9.7122 SD 9.3525 CD 4.6763 RF	166.7195	9.1951	42.343 SD 13.9419 SU 13.4665 SI 5.9344 CD 3.7671 DU
Static init blocks		17.5714	8.4286	43.9024 SI 22.7642 SD 4.065 SOC 4.065 DU 3.252 SPC	175.0127	9.2658	40.8723 SD 14.48 SU 13.4746 SI 7.0302 CD 3.7104 DU

Four classes with static non-final attributes were changed during bug-fix commits. The average number of changes is more than 3 time higher than for similar classes (286.5 vs. 80.9423). The number of modifications per commit is also greater (14.5 vs. 5.75). On the other hand, the classes that contain constants have low defect-proneness. Compared to other similar classes, their average number of modifications is less than half (72.75 vs. 166.561), but the number of changes per commit is higher (12 vs 9.061); this indicates that they were altered in much fewer bug-fix commits.

There are no stateful singletons in the latest version of Spring Core. Only 1 stateless singleton was modified during bug-fix commits (AnnotationAwareOrderComparator); it suffered 755 fine-grained changes over 30 commits, much more compared to other similar classes. The 15 utility classes that were fixed during the project's lifetime are less error-prone compared to the classes that were categorized as similar to them (in terms of size and complexity). The average number of changes is lower (122 vs. 170.6901) and the amount of modifications per commit is comparable (9.7333 vs. 9.0845).

Two classes that have static methods that access state were changed while fixing defects, AnnotationWriter and Frame. Both their average number of modifications and the amount of changes per commit are very high (699.5 vs. 138.8026 and 23.3734 vs. 8.7763, respectively). For the classes that contain static methods that solely operate on parameters it is the other way around. They have, on average, a smaller number of changes (69.5 vs. 166.7195) and were modified in a comparable number of commits (9.25 vs. 9.1951).

Similar to before, the classes with static initialization blocks have a much lower error-proneness. The average number of modifications is roughly 10 times smaller (17.5714 vs. 175.0127) and the number of changes per commits is close (8.4286 vs. 9.2658).

5.4.11. Wicket

Table 5.4.11.1: Change-proneness of classes with static constructs vs. similar classes for Wicket

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	128.1818	4.0909	23.4994 SI 18.0077 SD 15.0702 SU 7.9183 PTC 5.8748 DU	98.921	4.3267	16.7064 SI 16.4268 SD 14.8057 SU 5.8749 RF 5.517 DU
	Constants	35.5553	4.649	19.4783 SD 19.1396 SI 14.9545 SU 5.4778 RF	47.1797	3.793	16.352 SD 15.9215 SI 14.2077 SU 7.8324 DU

				5.0168 DU			7.435 RF
Singletons	Stateful	-	-	-	-	-	-
	Stateless	27.6667	4.3333	28.3133 SI 24.0964 SD 16.2651 SU 6.6265 SPC 3.6145 DU	89.9639	4.3238	17.6846 SD 17.4624 SI 15.807 SU 6.8635 RF 5.5271 DU
Static methods	Utility classes	92.7727	4.6818	22.44 SI 17.4914 SD 16.5605 SU 6.8594 RF 6.4674 DU	98.625	4.3117	18.7423 SD 18.3667 SI 14.7522 SU 5.8192 RF 5.4859 DU
	Access state	107.6	3.9852	18.8687 SD 15.6184 SU 14.3943 SI 7.5559 RF 5.0654 DU	97.0077	4.3364	19.6918 SD 19.6399 SI 13.7806 SU 5.7929 RF 5.5394 DU
	Operate on parameters	105.2571	5.2286	20.3995 SD 19.787 SU 15.3884 SI 5.4427 RF 5.3591 DU	86.4511	4.2156	18.1949 SI 18.1278 SD 15.7477 SU 5.9774 RF 4.5937 DU
Static init blocks		80.6667	4.6667	25.2066 SD 21.9008 SU 7.438 SI 5.3719 SPC 4.9587 RF	96.545	4.1213	18.4741 SD 18.5278 SI 14.5826 SU 5.66 RF 5.5248 DU

The 11 classes that contain static non-final attributes are more change-prone than classes which are similar to them in terms of size and complexity. There are more changes on average (128.1818 vs. 98.921) and the number of modifications per commit is roughly the same (4.0909 vs. 4.3267), thus indicating that they were altered in more commits. Four of the top 5 change types are identical, though the percentages are a bit different.

One the other hand, the classes with constants are not changed more frequently compared to similar classes. The average number of modifications is lower (35.5553 vs. 47.1797) while the amount of changes per commit is higher (4.649 vs. 3.793), therefore the number of commits in which they suffered modifications is also lower.

There are no stateful singletons in the final version of Wicket. Six of the 8 stateless singletons have suffered fine-grained source code changes throughout the project's lifespan. However, the average number of modifications is lower than for similar classes (27.6667 vs. 89.9639), while the number of changes per commit is almost the same (4.3333 vs. 4.3238); this would indicate that they are less change-prone. The top 5 change types also resemble the ones for similar classes; only the fourth in the rankings (STATEMENT_PARENT_CHANGE) is different.

Both the average number of changes (92.7727 vs. 98.625) and the number of changes per commit (4.6818 vs. 4.3117) for utility classes are very similar to those of other production classes. Additionally, the top 5 change types are also the same with statement-level modifications occurring most frequently. For the other classes that contain static methods, the average number of changes is higher (107.6 vs. 97.0077 for those with methods that access state and 105.2571 vs. 86.4511 for those with static methods that solely operate on parameters). However, for the ones from the first category the number of changes per commit is lower (3.9852 vs. 4.3364) than for similar classes, while for the classes from the second category it is higher (5.2286 vs. 4.2156). This indicates that the former were modified in more commits compared to other similar classes; this is not the case for the latter for which the number of commits is almost identical (roughly 20 commits). The top 5 change types are the same, just the order and percentages are slightly different.

The 3 classes with static initialization blocks, TagUtils (27 changes), WicketTagIdentifier (74), and JavaSerializer (141), have suffered less changes compared to similar classes (80.6667 vs. 96.545 on average). However, the number of changes per commit is comparable (4.6667 vs. 4.1213). The top 5 change types are also similar, but the percentages are quite different (e.g., 7.44% vs. 18.53% for statement inserts).

Table 5.4.11.2: Defect-proneness of classes with static constructs vs. similar classes for Wicket

Category		**Instances**			**Similar classes**		
		Avg. # changes	**# Changes per commit**	**% Occurrence to 5 change types**	**Avg. # changes**	**# Changes per commit**	**% Occurrence to 5 change types**
Static attributes	Non-final	55.4	3.9	28.7402 SD 24.8031 SI 12.9921 SU 4.7244 RF 4.3307 SPC	38.7883	4.3777	22.643 SD 19.745 SI 13.8784 SU 5.9654 SPC 5.1562 RF
	Constants	21.2228	4.7636	23.252 SD 20.1857 SI 14.3607 SU 5.7613 SPC 4.8647 RF	20.1762	3.6166	22.3164 SD 17.7966 SI 12.379 SU 6.8053 SPC 6.2917 RF
Singletons	Stateful	-	-	-	-	-	-
	Stateless	17.8	6.2	35.9551 SD 31.4607 SI 11.236 SPC 3.3708 CEC 3.3708 CD	38.6751	4.3538	21.6594 SD 20.7564 SI 13.927 SU 5.9227 SPC 5.162 RF
Static methods	Utility classes	35.4091	4.7273	24.9037 SI 22.8498 SD 13.9923 SU 7.3171 SPC 7.1887 DU	38.5911	4.3587	23.7001 SD 18.6561 SI 13.8619 SU 5.8906 SPC 5.144 RF
	Access state	76.5	4.7813	26.8489 SD 19.1318 SI 10.6109 SU 6.1093 RF 5.7879 SPC	38.6907	4.387	21.5913 SD 19.8248 SI 14.9712 SU 4.9494 RF 4.1213 SPC
	Operate on parameters	32.7368	3.8947	23.4263 SD 20.512 SI 14.2586 SU 5.3654 SPC 5.0908 RF	33.968	4.318	23.4152 SD 18.7127 SI 13.8248 SU 6.0822 SPC 5.2285 RF
Static init blocks		17	3	33.3333 SU 21.5686 SD 11.7647 SI 7.8431 ROS 5.8824 SPC	38.6043	4.3777	21.7171 SD 19.8239 SI 14.8278 SU 5.9448 SPC 5.1528 RF

Ten of the 11 classes with static non-final attributes were modified during bug-fix commits. Similar to change-proneness, they are more error-prone than other classes; the average number of changes is higher (55.4 vs. 38.7883) and there are approximately the same amount of changes per commit (3.9 vs. 4.3777). The top 5 change types are identical, but the order and percentages have small variations.

The classes that contain constants are not more error-prone when compared to similar classes. The average number of modifications is almost the same (21.2228 vs. 20.1762), but there is roughly 1 change more per commit (4.7636 vs. 3.6166); this implies that there are fewer bug-fix commits in which such classes were modified. Just as before, the top 5 change types are the same, even the percentages are very similar.

Five of the 8 stateless singletons were altered during bug-fix commits. The average number of changes is lower (17.8 vs. 38.6751) than for similar classes, but the amount of modifications per commit is higher (6.2 vs 4.3538); therefore, it is clear that they were modified in much fewer bug-fix commits. Another interesting observation is that statement updates were not performed on the stateless singletons during bug-fixing; the respective change type is not part of the ranking.

Bug fixes have been performed on 22 of the 31 utility classes; nevertheless, the defect-proneness for this kind of static constructs is exactly the same as for other similar classes. Both the average number of modifications (35.4091 vs. 38.5911) and the amount of changes per commit (4.7273 vs. 4.3587) are very close. The top 4 change types are also extremely similar (even percentage-wise).

Just as for change-proneness, the classes with static methods that access state are more error-prone than other similar classes while the ones that contain methods that only operate on parameters are not. For the first category the average number of changes is double (76.5 vs. 38.6907) and the amount of modifications per commit is almost the same (4.7813 vs. 4.387). Classes from the second category have suffered the same number of changes (32.7368 vs. 33.968) and the number of modifications per commit is comparable (3.8947 vs. 4.318). With regard to top 5 change types even the order is identical and the percentages are very close.

All the production classes with static initialization blocks were modified during bug-fix commits. However, the average number of changes performed is less the half (17 vs. 38.6043) while the amount of modifications per commits is only a bit less (3 vs. 4.3777); therefore, they were changed in fewer commits. Four of the top 5 change types are the same, but more statement updates have been done than for any other static constructs.

6. DISCUSSION

6.1. Revisiting the research questions

In the first section of this chapter we provide an interpretation of the results with regard to each research question. We look at the obtained results as a whole, thereby being able to draw meaningful conclusions. Below are our remarks per research question:

RQ1. Are static constructs used in complex software systems?

For the first research question we begin by establishing if instances from each category are present in the production code of the studied systems. In Table 6.1.1 we specify whether or not this is indeed the case per project; with 2 checkmarks we are representing that a considerable amount of instances of the respective type were found.

Table 6.1.1: Static construct presence

System	Static attributes		Singletons		Static methods			Static init. blocks
	Non-final	Constants	Stateful	Stateless	Utility classes	Access state	Operate on parameters	
BCEL	✓	✓✓	✓	✓	✓	✓	✓	✓
Commons Collections	×	✓✓	×	✓	✓✓	✓	✓	×
Commons Lang	✓	✓✓	×	✓	✓✓	✓	✓	✓
Commons Math	✓	✓✓	×	✓	✓	✓	✓	✓
Digester	×	✓✓	×	×	✓	×	✓	×
Geode	✓	✓✓	✓	✓	✓✓	✓	✓	✓
jHotDraw	✓	✓✓	×	✓	✓	✓	✓	✓
Pig	✓✓	✓✓	✓	✓	✓	✓	✓	✓
Spring Core	✓	✓✓	×	✓	✓✓	✓	✓	✓
Tomcat	✓	✓✓	✓	✓	✓✓	✓	✓	✓
Wicket	✓	✓✓	×	✓	✓	✓	✓	✓

All the projects contain static constructs, but not all categories of static constructs are present within a system. It can be observed that only 4 of the 11 systems (BCEL, Geode, Pig, and Tomcat) have instances from all 8 categories. While the last 3 are the largest projects studied, BCEL is considerably smaller; nevertheless, the project contains only 1 stateful singleton, the type that causes 5 of the other systems (except Commons Collections and Digester) not to appear in the previous list.

Static non-final attributes are encountered in 9 of the projects studied. Commons Collections and Digester are the only systems in which such instances are not present; they are 2 of the smallest projects in terms of size. There is also 1 system, Pig, which contains a considerable amount of static non-final attributes; thus, 2 checkmarks have been put in the corresponding entry in the table. The other type of static attributes, constants, are present all throughout the source code of the 11 projects that were analysed.

Ten of the projects contain at least 1 singleton. However, stateful instances are present only in the 4 systems enumerated above. On the other hand, stateless variants were found in all the projects except Digester (the smallest project studied).

Utility classes also appear in all the projects; the number of instances is substantial for 5 of them: Commons Collections, Commons Math, Geode, Spring Core, and Tomcat. Other types of static methods (that are not part of singletons or utility classes) have been encountered in all 11 systems. With the exception of Digester, static methods that access their class's state are present in the rest of the projects. Static methods that only operate on parameters have been found throughout the code. However, the amount of instances for both types is very low compared to the number of non-static methods.

Finally, static initialization blocks are present in 9 of the systems, but the number of instances is again on the low side.

Table 6.1.2: Percentage of instances per category

System	**Static attributes**		**Singletons**		**Static methods**			**Static init. blocks**
	Non-final	**Constants**	**Stateful**	**Stateless**	**Utility classes**	**Access state**	**Operate on parameters**	
BCEL	2.3121	80.2312	0.1156	0.2312	1.2717	6.474	8.7861	0.578
Commons Collections	0	52.5253	0	1.0101	6.2626	0.202	40	0
Commons Lang	0.125	65.625	0	0.125	6.375	0.375	25.5	1.875
Commons Math	1.3746	72.394	0	0.1145	2.8637	1.1455	20.7331	1.3746
Digester	0	80	0	0	4.4444	0	15.5556	0
Geode	2.8633	74.6491	0.158	0.595	2.8539	1.7105	16.7705	0.9947
jHotDraw	5.6054	82.7354	0	0.2242	0.8969	1.3453	8.7444	0.4484
Pig	12.1891	54.7761	0.6965	0.199	3.7313	4.2786	22.4378	1.6915
Spring Core	1.0363	63.3161	0	0.5181	6.5285	1.3472	24.456	2.7979
Tomcat	3.3779	78.2959	0.2017	0.126	3.756	1.1596	10.5873	2.4956
Wicket	2.3585	80.3235	0	0.5391	2.0889	0.4043	14.0836	0.2022

The table above shows the percentage of instances of a certain type from the total number of static constructs present within a system. Constants are by far the most common category; more than half of the static construct instances are constants for any of the projects. The lowest percentages are a bit above 50% (52.53% for Commons Collections and 54.78% for Pig), while the highest ones are over 80% (e.g., 80.32% for Wicket).

The second most common type of static constructs are static methods that only operate on parameters. This observation holds true for all the projects; nevertheless, the percentages differ considerably from one project to another; for BCEL and jHotDraw it is around 8.75%, while for Commons Collections the percentage is almost 5 times higher (40%).

The percentages for utility classes are generally higher than for the remaining types of static constructs. Similar to the previous category, there are cases in which the corresponding values are lower, such as jHotDraw (0.9%) or BCEL (1.27%).

An interesting observation can be made with regard to static non-final attributes. There are 2 projects, jHotDraw and Pig, in which the percentages for this type of static constructs are considerably higher (5.61% and 12.19%, respectively) than for the previous category. Nonetheless, there are more cases where they are much lower, including Commons Collections, Commons Lang, Digester, or Spring Core (under 1%).

Static methods that access state are next in this ranking; however, this is the case for only some of the systems, such as BCEL, Geode, jHotDraw, or Pig. There are also situations in which the percentage is 0 (Digester) or very small (0.202% for Commons Collections).

There are more instances of static initialization blocks than singletons (regardless of their kind). The only projects that do not adhere to this rule are Commons Collections and Wicket. There is also Digester which does not have instances of any of these types.

Finally, stateless singletons appear in 10 of the 11 systems while the stateful ones are present in only 4 of them. From these 4 projects, the percentages for the stateful variant are higher in 2 of them (Pig and Tomcat) and lower in the other 2 (BCEL and Geode). In terms of actual types, Eager Instantiation seems to be the predominant type followed closely by the general form (Lazy Instantiation). Other variations, such as Subclassed Singleton or Limiton, were rarely found in the studied systems; however, it is worth mentioning that in Geode (the project with the highest amount of instances) there are 2 hierarchies in which most of the classes are Subclassed Singletons.

RQ2. How have static constructs evolved throughout the lifespan of a project?

For the second research question we analyse each category of static constructs separately in terms of evolution. We compare the percentage of instances of a particular type for the initial version of a project and the latest one studied. Additionally, the maximum value for this percentage along with the date in which it was reached are also recorded. These measurements allow us to determine whether or not the number of instances increased as a system grew in size or if they are utilized less nowadays.

Table 6.1.3: Evolution of static attributes and singletons

System	Constants			Static non-final attributes			Stateful singletons			Stateless singletons		
	First	Max.	Last	First	Max.	Last	First	Max.	Last	First	Max.	Last
BCEL	2.66	3.43 5/2003	1.17	54.68	70.33 9/2015	40.75	0.56	0.56 11/2001	0.23	0.56	0.84 9/2002	0.46
Commons Collections	2.86	5.69 2/2002	0	2.86	34.24 8/2012	29.85	0	0	0	0	1.66 5/2013	0.95
Commons Lang	3.39	6.15 5/2003	0.001	64.41	67.08 12/2009	60.57	0	0	0	0	0.39 2/2020	0.31
Commons Math	0	9.17 9/2007	0.56	17.33	34.46 1/2016	29.74	0	0	0	0	0.45 2/2007	0.12
Digester	1.49	9.91 5/2004	0	1.49	10.91 11/2011	10.91	0	0	0	0	0.66 8/2010	0

Geode	2.41	2.41 4/2015	1.39	53.18	53.18 4/2015	36.21	0.53	0.53 5/2015	0.44	2.09	2.12 6/2015	1.41
jHotDraw	3.55	17.86 5/2009	2.21	20.49	42.91 5/2020	42.91	0.56	0.56 9/2000	0	0	0.34 5/2020	0.34
Pig	8.06	13.24 12/2008	6	16.12	25.36 11/2014	24.58	1.13	1.32 2/2008	0.8	0	0.36 3/2008	0.23
Spring Core	2.22	6.77 8/2010	0.52	41.78	41.78 10/2007	31.97	0	0.25 7/2013	0	1.05	1.15 12/2013	0.77
Tomcat	7.5	7.5 2/2006	1.39	28.14	33.76 11/2010	32.18	0.48	0.48 3/2006	0.38	0.1	0.24 12/2020	0.24
Wicket	1.37	1.79 10/2009	0.84	44.77	45.06 12/2008	39.59	0.09	0.09 3/2007	0	0.99	1.18 1/2012	0.65

For static non-final attributes we calculated the percentage of instances from the total number of attributes and the situation is straightforward. This percentage is higher in the first version of a project compared to the latest one. The only system that does not adhere to this rules is Commons Math, because there were no static non-final attributes in its initial version; however, the percentage for the last version is also very low (0.56%). Furthermore, the maximum values for this percentage were reached towards the beginning of the development process; the latest maximum was encountered in 2015 (but it corresponds the first version of Geode available). This indicates that the developers have become aware of the problems caused by static non-final attributes and started to utilize them less.

On the other hand, for constants no clear pattern could be observed. For 6 of the systems the percentage of constants from all the attributes is higher in the latest version, while for the other 5 it is greater in the initial one. There are many cases in which the percentages are very close, for example Commons Lang, Pig, Tomcat, or Wicket. In general, it was observed that the number of instances increased proportionally to the total number of attributes. In terms of the maximum values, they were encountered around halfway through the development period in most cases. The only exception would be jHotDraw; for this system the peak percentage was found for the last version studied.

The percentage of singletons from the total number of production classes is very low throughout the lifespan of any system; thus, the idea that singletons are overused is not supported by the obtained results. For the stateful variant, there are 4 projects in which there were no such instances throughout their entire existence and 3 with no singletons of this type in their latest version. Even for the other 4 systems, the percentage of stateful singletons is extremely low (less than 1%). Additionally, the maximum values for this percentage were encountered at the beginning of the development process in all cases. Stateless singletons were utilized a bit more frequently. Although they appeared in the initial version of only 4 projects, Digester is the sole system that does not have such instances in its final version. The percentages are again low, but they are a little higher than for stateful singletons. There are also situations in which the maximum was found towards the end of the development cycle, such as Commons Lang, jHotDraw, and Tomcat. This suggests that the developers are not as reluctant to create stateless singletons compared to their stateful counterparts.

Table 6.1.4: Evolution of utility classes, static methods, and initialization blocks

System	Utility classes			Methods access state			Methods only parameters			Static init. blocks		
	First	Max.	Last	First	Max.	Last	First	Max.	Last	First	Max.	Last
BCEL	1.67	2.55 4/2021	2.55	0.43	0.63 6/2006	0.35	2.23	3.31 6/2019	3.17	4	5 8/2015	5
Commons Collections	11.11	11.11 5/2001	5.9	0	0.28 2/2002	0.02	0.35	5.16 9/2007	4.45	1	2 5/2013	0
Commons Lang	42.86	42.86 8/2002	16.04	0	1.69 2/2004	0.08	0.29	5.7 4/2020	5.67	2	16 5/2008	15
Commons Math	5.71	10.53 6/2003	3.04	0	0.91 11/2008	0.17	0.27	3.86 7/2016	3.12	0	19 3/2016	12
Digester	0	6.7 1/2004	1.06	0	0.35 11/2003	0	0.57	0.84 8/2010	0.76	0	1 1/2004	0
Geode	5.79	5.79 5/2015	5.32	0.67	0.67 4/2015	0.33	3.79	3.79 4/2015	3.24	171	171 4/2015	107
jHotDraw	1.69	5.01 9/2003	1.03	0.79	1.94 1/2010	0.22	1.44	3.94 1/2003	1.43	0	23 11/2010	8
Pig	2.82	6.29 4/2008	4.1	0.79	0.96 1/2018	0.95	2.82	5.11 1/2017	4.98	0	39 10/2015	34
Spring Core	0	19.16 10/2008	9.75	0	0.44 5/2018	0.27	0.43	4.98 9/2018	4.89	7	28 12/2015	27
Tomcat	7.13	7.41 4/2006	7.01	1.85	1.85 3/2006	0.22	4.13	4.13 3/2006	1.98	49	110 10/2018	99
Wicket	1.53	2.64 3/2016	2.02	0.23	0.33 3/2008	0.07	2.69	3.18 10/2009	2.28	0	4 11/2015	4

For utility classes, if there were such instances in the first version of a system, then their percentage is higher than the corresponding value for the last version studied. This is true for 6 of the 9 projects in this situation; for all 3 remaining ones, BCEL, Pig, and Wicket, the percentages for the initial and final versions are very close. Similar to before, the maximums appeared at the start of the development period. There are nonetheless exceptions, such as BCEL or Wicket. The fact that utility classes are used less in recent years is surprising especially for the projects that are structured as libraries (which rely heavily on such static constructs).

For static methods that access state, the situation is similar to the one described above. If the percentage is greater than 0 in the initial version, then it is also higher than the value obtained for the latest version. Pig is the only exception, but for this project the values are relatively constant throughout its entire existence. For the projects which had no instances of this type initially, the percentage increased considerably in the first few months of development and eventually became higher than that of the latest version. In general, the maximums were encountered at the very beginning of the development process, proving once again that there are some types of static constructs that are being used less and less.

The case for static methods that only operate on parameter is very different. For 7 of the projects the percentage of instances is higher in the latest version analysed, while for 3 of the remaining ones the values are close (e.g., for jHotDraw). This is especially true for some of the projects, such as Commons Collections or Commons Math, where the difference is substantial. Furthermore, the maximum values were reached towards the end of the development cycle for more than half of the systems.

For static initialization blocks we reason in terms of number of instances. We could have provided the percentage of production classes that contain such instances instead, but decided not to due to the fact that a class may have 2 or more static initialization blocks. It can be observed that for 8 of the 11 systems the number of instances is higher in the latest version compared to the initial one. For Commons Collections and Digester the amount of static initialization blocks is low throughout their entire lifespan. However, for Geode it dropped considerably even though the number of production classes only decreased from 4992 to 4528. Another observation would be that the maximum number of instances was generally encountered halfway through the development process. There is only 1 system, Tomcat, for which it was found towards the end (in late 2018).

RQ3. Do static constructs have a negative impact on software quality aspects?

We evaluate the impact of each category of static constructs on the 3 quality aspects considered. The types are studied independently in order to establish which of them are the most harmful with respect to a certain aspect.

Table 6.1.5: Impact of static constructs on class testability

System	Static attributes		Singletons		Static methods			Static init. blocks
	Non-final	Constants	Stateful	Stateless	Utility classes	Access state	Operate on parameters	
BCEL	<<	>	<	-	>	≈	>>	>>
Commons Collections	-	<<	-	>	≈	≈	≈	-
Commons Lang	<<	>	-	>>	<	<	>	>
Commons Math	<<	<	-	<	≈	<	>	<<
Digester	-	>	-	-	≈	-	≈	-
Geode	<<	>	≈	>	>>	≈	>	>
jHotDraw	-	-	-	-	-	-	-	-
Pig	<<	≈	<	>	≈	<<	>	≈
Spring Core	<<	<	-	>>	≈	<	>>	≈
Tomcat	>	>>	≈	≈	>	>	≈	<
Wicket	<<	≈	-	>	>>	<	>>	≈

In the above table we provide an overview of the testability of the classes that contain different types of static constructs when compared to similar classes. Two symbols (<< or >>) are used to indicate that both the quantitative and the qualitative scores are higher in favour of one or the other. Only 1 symbol (< or >) shows that although the 2 scores differ (one is greater while the other is lower), the overall testability score is still considerably higher either for the classes of interest or for the similar classes.

For the ones with static non-final attributes it is clear that they are less testable than the classes which are similar to them in terms of size and complexity. From the 8 systems in which such instances appear, for 7 of them the difference is heavily in favour of the similar classes. There is only 1 exception, Tomcat, but even for this project the classes that contain static non-final attributes are only tested more (the unit tests are not of a better quality). It is worth mentioning that for some systems the results might be skewed due to the small number of classes that contain this kind of attributes (e.g., Commons Lang with 1 such instance).

The situation is not as straightforward for the classes with constant. For the 10 systems analysed in terms of testability we found that: in 5 of them the classes of interest have a higher overall testability score (especially in Tomcat), for 2 (Pig and Wicket) the scores are roughly the same, while for the other 3 the similar classes are more testable. There are numerous production classes that contain constants, therefore most of the remaining classes were included in the group of similar classes; this might cause the results to be more general than for the other categories of static constructs.

Only 4 systems have stateful singletons in their latest versions. For 2 of them the overall testability of the singletons is lower than that of similar classes, while for the other 2 the values are more or less the same. On the other hand, the stateless variants appear to be much more testable than their stateful counterparts. There are 8 projects in which instances of this type exist and for 6 of them their corresponding score is greater than for similar classes. In 2 of the cases, Commons Lang and Spring Core, both the quantitative and the qualitative scores are higher; however, the number of instances in these systems is quite low (1 and 5 stateless singletons, respectively). For Tomcat the testability scores are very similar, while for Commons Math (only 1 instance) they are in favour of the similar classes.

The observations for utility classes resemble the previous ones (for the stateless singletons), albeit the number of instances is much higher. There are 4 systems in which the utility classes are more testable than other similar classes, 5 where the overall testability scores are comparable, and only 1 (Commons Lang) that does not adhere to the rule. For 2 of the projects from the first category, Geode and Wicket, the difference is substantial in favour of the utility classes.

For the rest of the production classes that contain static methods, the cases for those with methods that access their state and for the ones with static methods that only operate on parameters seem to be the opposite of one another. There are 5 out of 9 projects for which the classes with static methods that access state are less testable. From the remaining 4 only Tomcat is an actual exception, in the other 3 (BCEL, Commons Collections, and Geode) the overall testability scores are very similar. For static methods that solely operate on parameters, the classes that contain them have a higher testability in 7 of the 10 cases; in 3 of them (BCEL, Spring Core, and Wicket) both scores are greater. For the remaining 3 systems, the overall testability scores for the classes on interest and the similar classes are comparable; there is no situation in which the testability is higher for the latter category.

Finally, the classes with static initialization blocks also appear to be a bit more testable. Instances of this type are present in 9 of the projects and the cases are as follows: for 4 of them (especially for BCEL) the classes of interest have a higher testability, in 3 the overall testability scores are roughly the same, while 2 of the systems represent exceptions. For Commons Math there are 12 instances of static initialization blocks and the classes that contain them have a much lower testability score compared to similar classes; this is mainly due to the large difference in terms of coverage, thereby causing their quantitative score to be significantly smaller.

Table 6.1.6: Impact of static constructs on change-proneness

System	Static attributes		Singletons		Static methods			Static init. blocks
	Non-final	Constants	Stateful	Stateless	Utility classes	Access state	Operate on parameters	
BCEL	>>	≈	>>	<<	>>	>>	>>	>>

Commons Collections	-	≈	-	<<	>>	≈	<	<
Commons Lang	≈	≈	-	-	>>	>	<<	>
Commons Math	≈	<	-	-	>>	>>	<	>>
Digester	-	≈	-	-	<<	-	>	-
Geode	>>	<<	>>	>>	≈	>>	>>	>>
jHotDraw	≈	>	-	-	-	<	<<	>>
Pig	>	>	>	<<	≈	>>	>	>>
Spring Core	≈	<<	-	>>	<<	>>	<<	<<
Tomcat	>>	<	>	>	<	>>	<<	≈
Wicket	>	<	-	<<	≈	≈	≈	≈

Table 6.1.6 presents the change-proneness of the classes with different types of static constructs in comparison to that of similar classes. The symbols are the same as for the previous table; however, in this case 2 symbols are used to represent that both the average number of changes and the number of commit in which the instances from a certain category were modified are higher / lower.

The classes that contain static methods that access state have the highest change-proneness when compared to other similar classes. There are 10 systems in which such instances appear and for 7 of them this is clearly the case. Even for 2 of the others, Commons Collections and Wicket, the change-proneness of these classes is comparable to that of similar classes; they are by no means less susceptible to modifications. The only exception is jHotDraw, but for this system only 1 such instance was encountered. In general, the difference between the average number of changes is substantial while the amount of modifications per commit in not.

For all 4 projects that have stateful singletons we found that the respective instances are more change-prone than the classes that were categorized as similar to them (in terms of size and complexity). Especially for 2 of the systems, BCEL and Geode, the average number of modifications is much higher while the number of changes per commit is lower; this implies that the stateful singletons were changed in many more commits. The latter (Geode) is the project with the most instances of this type out of all the systems investigated.

Another category that resembles the previous ones are classes with static non-final attributes; their change-proneness is also greater than that of similar classes. Nine of the 11 projects have such classes and the situation is as follows: for 5 of them the respective classes are more susceptible to changes, while in 4 the change-proneness is comparable.

Classes that contain static initialization blocks are also more change-prone. Six of the 10 projects with such instances adhere to this rules, while for 2 of the others (Tomcat and Wicket) the values are very close. The exceptions are Commons Collections (only 1 instance) and Spring Core, a system which does have a significant number of classes with static initialization blocks (25); 17 of them have been modified throughout its history.

An interesting case is that of utility classes; it would seem that they are more prone to modifications in projects that are structured as libraries, while for the other systems it is the other way around. For the 3 Commons libraries and BCEL the utility classes have a higher change-proneness, which might indicate that a special emphasis is put on these classes in this kind of projects.

Statefull singletons are in a similar situation; there are 7 projects in which these singletons have suffered modifications throughout their lifespan. In 3 of them the average number of changes is considerably higher compared to similar classes, while for the other 4 this is not the case; there is no system for which the values are close. It is also worth mentioning that projects with a small number of instances appear in both categories (e.g., Spring Core for the first category and Commons Collections for the latter).

Finally, the classes that contain constants / static methods that only operate on parameters are not more change-prone than other similar classes. Instances of both types are present in all of the 11 projects. For the former category there are: 5 systems (especially Geode and Spring Core) in which the classes with constants are less change-prone than similar classes, 4 where the proneness is relatively the same, and 2 in which it is lower. For classes with static methods that solely operate on parameters the situation is more polarized: there is only 1 project in which the values are close, for 6 of the others the classes of interest have a lower change-proneness, while for the remaining 4 systems they are more susceptible to modifications.

Table 6.1.7: Impact of static constructs on defect-proneness

System	Static attributes		Singletons		Static methods			Static init. blocks
	Non-final	Constants	Stateful	Stateless	Utility classes	Access state	Operate on parameters	
BCEL	>	>	≈	<<	>	>>	≈	>
Commons Collections	-	≈	-	-	>>	≈	<	-
Commons Lang	>	<	-	-	>>	>	<	≈
Commons Math	≈	≈	-	-	>>	>	≈	>>
Digester	-	≈	-	-	-	-	≈	-
Geode	>>	≈	>	>>	≈	>>	>	>>
jHotDraw	≈	>	-	-	-	≈	<<	>>
Pig	>	>	>	<<	>	>	<	>
Spring Core	>>	<<	-	>>	<	>>	<<	<<
Tomcat	>	<	>	≈	<	>>	<	<
Wicket	>	≈	-	<<	≈	>>	≈	<<

Out of all the classes with static constructs, the ones that contain static methods that access state are by far the most defect-prone. Instances of this type are present in 10 of the 11 systems and in 8 of the cases their error-proneness is higher than that of similar classes; for 5 of the projects the difference is substantial. There are only 2 systems in which the values are comparable, Commons Collections and jHotDraw, but even for them the average number of changes is greater in favour of the classes with this kind of static methods.

Classes that contain static non-final attributes are almost as defect-prone as the previous ones. This observation holds true for 7 (especially Geode and Spring Core) out of the 9 projects in which such instances appear. The only exceptions are Commons Math and jHotDraw; for them the average number of modifications in the classes of interest / similar classes are very close.

Stateful singletons are also more error-prone than other similar classes, but the difference is lower than for change-proneness. Three of the 4 systems adhere to this rule, while for BCEL the corresponding values are almost identical. Just as for change-proneness, the results for the stateless variant are inconclusive; for 2 of the projects the instances have a higher defect-proneness when compared to similar classes, for 3 of the remaining ones it is the other way around, while for Tomcat the average number of changes and the amount of modifications per commit are more or less the same.

For utility classes the situation is as follows: in 5 of the projects the defect-proneness is higher than that of similar classes, for 2 (Geode and Wicket) it is comparable, and in the last 2 systems (Spring Core and Tomcat) the instance of interest are less prone to error. This is quite different compared to what was observed for change-proneness; there the utility classes were more prone to modifications only in the systems that were structured as libraries.

Classes with static initialization blocks are in a similar situation. Instances were changed during bug-fix commits in 9 of the 11 projects and for 5 of the systems the respective classes are more defect-prone. For 3 of the largest projects, Spring Core, Tomcat, and Wicket, this is not the case, while for Commons Lang the ratio between the average number of modifications and the amount of changes per commit is close to the one obtained for similar classes.

Like for change-proneness, the classes that contain constants and static methods that solely operate on parameters also appear to be less error-prone. This is very clear for the latter category where there is only 1 exception, Geode. In 6 of the projects (especially Spring Core) the classes with this type of static methods have a lower defect-proneness than their similar counterparts, while for the other 4 systems the computed values are comparable. Finally, for the classes with constants there are: 5 systems for which the values are close, 3 (Commons Lang, Spring Core, and Tomcat) where their error-proneness is lower, and 3 (BCEL, jHotDraw, and Pig) in which they are actually more susceptible to defects than other similar classes.

6.2. Threats to validity

In this section we present the factors that could be considered threats to the validity of the empirical study and the obtained results. Additionally, we discuss the ways in which we tried to mitigate them. The factors are grouped into 3 categories: construction, internal, and external threats. This categorization is done based on the guidelines introduced by *Perry et al.* in [98]. The threats from the first category are related to the independent and dependent variables; more specifically, whether or not they model the formulated hypotheses accurately. The internal threats arise if the changes in the dependent variables cannot be attributed to changes in the independent ones. Finally, the threats from the third category address the results of the empirical study, namely if they are generalizable to other settings. For each category we have identified a series of threats:

1. **Construction threats**
   These threats may appear due to problems in the code that was developed for collecting the data required in our analyses. To avoid such issues, the proposed approach was carefully tested using several small-scale systems created specifically for this purpose. For static construct presence / usage we added instances from each category in different combinations and checked that they are detected correctly. As an example, for singletons we manually recreated all the variations discussed in [91] while also studying jHotDraw with commits from the time of the article. Furthermore, we verified the number of clients for each instance to determine if they are calculated correctly along with their localization. In terms of evolution, we randomly chose commits from every system and checked that the number of instances / clients of each type are identified properly. For testability we

ensured that the quantitative and qualitative metrics are computed correctly; the coverage and test smell data was inspected to confirm that 1) the percentages of production methods addressed by tests / unit tests that contain smells and 2) the number of different types of smells from a test class are in order. We also corroborated that the corresponding scores are determined properly based on the proposed threshold values. Finally, for change- / defect-proneness we verified that the fine-grained source code changes were extracted correctly by manually comparing consecutive commits to establish what was modified. Additionally, we checked 1) that the lists of issue keys corresponding to bugs are correct and complete and 2) that the bug-fix commits are identified accurately.

2. **Internal threats**
The main threats from this category are confounding factors, namely other variables that could mask an actual association or falsely prove an apparent association between the independent and dependent variables. It is difficult to identify all the factors that have an impact on these variables, but we will try to discuss as many as possible. For the first hypothesis, there might be other system characteristics that affect the presence / usage of different types of static constructs. Studying multiple combinations of characteristics can help alleviate this threat. With regard to the second hypothesis, there may be other factors that influence the evolution of static constructs. Though they are important, the size and complexity of the studied systems should not be the sole causes why instances / their clients were created or deleted. Finally, for the last hypothesis, the presence of certain types of static constructs might not be the only reason why a class suffers from lack of testability or has high change- / defect-proneness. This is especially true for the smaller constructs, such as constants or static methods that are not part of singletons / utility classes; their impact on the 3 software quality aspect should be lower than for the other categories.

3. **External threats**

- Only open-source projects: one of the biggest threats from this category is that the observations presented above might not be generalizable for other software systems. Up until this point, we did not have access to projects from industry, therefore all 11 systems that were included in the analysis are open-source. Different types of static constructs might appear more frequently in commercial projects and they could also be utilized in other ways. Furthermore, their evolutionary patterns may not resemble the ones that have been observed thus far. The way in which class testability is assessed (in relation to other similar classes) could be inappropriate if all the production classes are fully tested with unit tests of the highest quality. Finally, change- and fault-proneness might prove easier to quantify due to better development practices (e.g., more fine-grained commits or better commit messages). All in all, there is a clear need to study industrial projects; we are actively working on addressing this threat and expect to obtain access to at least 2 commercial systems in the near future.
- Only object-oriented Java systems: another cause for concern may be that all the projects included in the empirical study were developed in Java by following an object-oriented approach. In terms of programming language, we are confident that the solution can be easily adapted to enable us to analyse systems created in other object-oriented languages (such as C# or C++). The code might need to be reimplemented in the respective

language, but the proposed approach should still apply. However, studying systems which are developed following other programming paradigms could prove more difficult. For example, class testability is investigated in an object-oriented context, by evaluating the quantity and quality of the associated unit tests. This may not be possible for projects created in imperative or functional programming languages because some of the concepts on which our approach is based might not exist. We need to experiment with other programming languages / paradigms before we can generalize the obtained results.

- High granularity of study: the level at which everything is studied could also be considered a threat to validity; most of the analyses are performed at class level. Although there are situations in which an analysis is more fine-grained (e.g., the extracted source code changes), in most cases we only look at classes as a whole. For example, we categorize a production class as a singleton client if at least 1 of its methods utilizes the singleton. Studying exactly which methods use it would have allowed for a better categorization, thereby obtaining more detailed results. Granularity may also impact the process through which we study the evolution of static constructs. In our approach we performed sampling on a system's commits with a frequency of 1 per month. The results could have been a bit different if all the commits were considered. As explained in Chapter 3, the probability that a static construct was added and immediately removed within that 1 month period is quite low. Nonetheless, we can extend the approach so that 1) the projects are studied at method level or 2) all the commits are analysed.
- System selection process: the projects that were chosen for the empirical study could also represent an external threat. Even though they were selected based on a set of well-established criteria, there may be other systems with completely different characteristics that should have been included in the analysis. We tried to choose projects 1) of various sizes and complexities, 2) with unique development practices, and 3) varying testing efforts. However, systems created by following certain development methodologies might be worth considering. While the specific methodology may not influence all the software quality aspects, there could be some that are impacted. For example, we expect Test Driven Development to affect our assessment on class testability. The code coverage for projects developed following TDD should be significantly higher because the corresponding unit tests must be written before the production classes are implemented. On the other hand, this does not guarantee that the quality of the tests will be higher. In the same vein, the change- / defect-proneness of the production classes should be lower because the requirements are clearer and less bugs are introduced. The above are just suppositions, we need to analyse this kind of projects before any meaningful conclusions can be drawn.

To conclude this subsection, we would like to reiterate that we selected systems with different characteristics, a considerable number of versions were analysed, and both the test suites and the change histories were appropriate. Although there are several ways in which the empirical study can be improved, we firmly believe that it represents a solid foundation for the research that will follow. We plan to address all the aforementioned external threats in the near future, as will be explained in the Future Work section.

In summary, this chapter discusses:

1. The implications of the results with regard to each research question:
   - For **RQ1** we found that: a) static constructs are present in all the systems studied; b) constants are by far the most common type followed by static methods and utility classes; static non-final attributes, initialization blocks, and singletons appear less often, especially in the smaller projects; c) the number of clients of static constructs (and their localization) are not much different than those of similar classes in most of the cases.
   - For **RQ2** we saw that there are indeed several categories of static constructs for which fewer instances are added nowadays compared to earlier stages of development.
   - For **RQ3** we established that: a) static non-final attributes, stateful singletons, and static methods that access state have the highest impact on class testability; b) all the categories of static constructs except constants and static methods that only operate on parameters affect change-proneness, albeit for utility classes and stateless singletons the results are contradictory for different types of systems; c) for defect-proneness their impact is not as significant as for change-proneness.
2. The threats to the validity of the empirical study:
   - construction threats: problems in the code that was developed in order to a) detect static constructs, b) study the evolution of different types of instances and their clients, and c) quantify class testability, change- and defect-proneness.
   - internal threats: lack of / wrong correlation between the independent and dependent variables for each hypothesis.
   - external threats: lack of generalizability of the results because a) all the projects are open-source and implemented in Java, b) the granularity of the study is too high, c) the system selection process was inappropriate.

7. CONCLUSIONS

Developing software systems is a complex process which is comprised of several activities, including: design, implementation, testing, and deployment. Performing these activities can be difficult if a project does not possess some key quality features. For example, testing could be hindered because the system's production classes lack testability. The implementation time might also increase due to the high change- / defect-proneness of the classes. These aspects need to be taken in to account especially in the context of evolution. As projects evolve, they still need to meet a series of quality requirements, such as: performance criteria (e.g., speed or accuracy), being easily maintainable and testable, or not being susceptible to change / defects. However, little research has been done thus far on what makes a system 1) difficult to test and 2) change- / 3) error-prone. Static constructs have already been shown to have a negative effect on understandability, maintainability, and efficiency, but there are other aspects that still need to be studied. We have been addressing this knowledge gap by conducting an empirical study which investigates the impact of static constructs on the quality aspects mentioned above.

First, we categorized the static constructs and defined detection strategies through which instances of each type can be identified. Afterwards we studied these instances both for the latest version of a system and throughout its entire lifespan; this was done to determine how static constructs have evolved over time. Finally, we defined models that can be used to quantify the 3 quality aspects investigated. For a part of the production code that contains / utilizes static constructs we can determine if it is less testable or more change- / defect-prone compared to other similar classes.

In this final chapter of the thesis we start by providing an overview of the main scientific contributions made through our work. Then we summarize the results that were obtained and discuss the conclusions that can be drawn from them. Next, we reflect on what we have accomplished and explain what could have been done better. We end the chapter with future work directions that we are currently pondering.

7.1. Contributions

In this thesis we investigate static constructs, how they evolved and their effect on various software quality aspects. This was done in order to: 1) determine how they are currently utilized, 2) compare this to the way in which they were used throughout the lifespan of a system, 3) establish if they have a negative impact on testability or change- / defect-proneness. By doing this we bring the following contributions:

1. A methodology for studying the evolution and the impact on 1) testability and 2) change- / 3) defect-proneness of any design flaw. Even though in the thesis we focus on static constructs, the proposed approach can be used to investigate other design flaws. The corresponding detection strategies need

to be defined, then the evolution and the effect on software quality may be studied in a similar manner. For example, the God Class design flaw can be detected as proposed in [33], by computing the corresponding metrics WMC (Weighted Method per Class), Tight Class Cohesion (TCC), and Access to Foreign Data (AFD) and comparing them to the threshold values. The results are combined into a detection strategy which can be utilized to identify instances of the God Class flaw. The presence of such instances can be analysed both for the latest version of a project and for its entire history. Finally, their impact on the software quality aspects can be established by making use of the proposed quantification models. As an example, we can determine if God Classes are changed more frequently during bug-fixing commits compared to the rest of the production classes.

2. A model for quantifying the testability of a production class. Unlike other publications that address software testability, we evaluate this quality aspect based on the test code rather than the production code. We consider that a part of the system is tested less / with lower quality unit tests compared to other parts of the code because it is more difficult to test (has low testability). Therefore, testability was evaluated both from a quantitative and from a qualitative perspective. In terms of quantity, we relied on code coverage data; for quality we determined if particular smells are present in the associated unit tests. These 2 aspects were combined in order to compute a testability score for a specific part of the production code.
3. A process for determining 1) what was changed during a commit and 2) whether or not that particular commit is a bug-fix. First and foremost, to evaluate change-proneness we needed to be able to establish the exact modifications that were made during a commit. For this we extract fine-grained source code changes which specify: the entity that was modified (class, attribute, or method), the type of the change (e.g., conditional statement modification in method), and other details related to it (such as severity). We use these data to determine if a class that has / utilizes static constructs is more change-prone than other production classes. The entire change history of the studied class is analysed and compared to that of similar classes (in terms of size and complexity). Defect-proneness is evaluated in the same manner, but only the commits that were categorized as bug-fixes are taken into account. In order to determine if a commit is a bug-fix we rely on 2 types of information: 1) the one available in the commit message and 2) additional data collected from the corresponding Jira issue tracker. Based on this we are able to accurately categorize commits as bug-fixes.
4. A tool that incorporates all of these aspects. The aspects that were discussed above were integrated into DFAnalyser. This tool is an extension of Patrools [95] which could already compute some of the required metrics. We designed it to have a modular structure; several modules can be combined together in order to perform the required analysis. One of the modules contains the detections strategies for the design flaw that is being investigated (e.g., singletons). Another module is concerned with the software quality aspect for which we want to assess the impact of the design flaw. Finally, if we also want to study the evolution of the respective flaw, we have to add the appropriate module. The modules are highly configurable and can be easily extended; for example, a different quality

aspect could be investigated by creating a module with a suitable model for quantifying it.

5. An empirical study through which we answer the proposed research questions. We began our research by formulating a series of questions that cover the major aspects that we wanted to understand with regards to static constructs. For each of the research questions we also prepared several hypotheses that we needed to test. To answer them we conducted an empirical study which includes 10 open-source software projects. In this study we investigate all the aspects discussed above, namely: static construct usage, evolution, impact on testability and change- / defect-proneness. Each type of static construct was analysed in isolation; afterwards, some general conclusions have been drawn for static constructs as a whole.
6. A better understanding of static constructs, their evolution, and the effect they have on various software quality aspects. First, we wanted to establish if static constructs (e.g., mutable global state) are present in the production classes of complex software projects and whether or not other classes utilize them. Then we were keen to observe how they are used nowadays, once a system has reached maturity, compared to earlier stages of development. Finally, we assessed the effect of static construct usage on quality aspects such as testability or change- / defect-proneness; we determined which types of static constructs have the biggest negative impact on the aforementioned quality aspects and discussed possible reasons why this is the case.

7.2. Conclusions

The proposed approach was successfully implemented and an empirical study which includes 10 open-source systems was conducted. Some interesting findings were obtained through this study. We are now capable of answering the research questions that were formulated:

For **RQ1**, "Are static constructs used in complex software systems?", we analysed the presence and usage of each type of static construct for the latest version of a project. The main finding is that instances of static constructs actually do appear in the code and are frequently utilized by other production classes. Classes with static non-final attributes, constants and static methods are present all throughout the source code, while singletons and static initialization blocks are used, but to a smaller extent. We make a distinction between stateful and stateless singletons; those from the latter category seem to appear more often. We also divided the static methods which are not part of singletons into 3 categories: 1) those from utility classes; 2) that utilize the attributes of their class; 3) which only operate on parameters. Based on the specific characteristics of a system, static methods from one category are used more compared to the others. For example, in a project which is structured as a library (Commons Math), the most common type of static methods are the ones that are part of utility classes.

In terms of usage, the number of production classes that utilize such instances varies depending on the static construct's type. As an example, there are more classes that use static methods (regardless of their category) than there are singleton clients. Unlike the other types, static non-final attributes, constants, and static initialization blocks are generally utilized within the class in which they appear rather than from other production classes.

For **RQ2**, “How have static constructs evolved throughout the lifespan of a project?”, we studied the evolution of each type of static construct in isolation for monthly versions of a system. In general, it was observed that most of the static constructs are utilized less once a project reaches maturity. The percentage of instances present from each category is usually higher in the initial versions of a system compared to the latest ones. Also, the maximum number of instances of a particular type was encountered more frequently towards the beginning of the development process. For example, there were more singletons in a version that is halfway during the development period than in the final version studied although the number of production classes is constantly growing. The only exceptions are constants and static methods which solely operate on parameters (to some extent); for these 2 categories the amount of instances increases continuously as a project grows in size.

The situation is even more evident for static construct clients. There are numerous cases in which the number of clients for a particular instance remained constant (or even decreased) while the total number of production classes for the system was growing exponentially. Situations in which the classes with / that utilize static constructs were marked as Deprecated have also been encountered. Starting from that version, the number of client classes began to decrease until reaching 0 (or until the respective class was removed). All of the above suggest that the developers have become aware of the problems associated with the usage of specific types of static constructs and started to utilize them less.

For **RQ3**, “Do static constructs have a negative impact on software quality aspects?”, we investigated the effects of each category of static constructs on the 3 quality aspects addressed by this study. For testability we found that some of the instances have a more detrimental effect compared to others. Stateful singletons and static non-final attributes appear to have the biggest impact on the testability of the production classes that utilize them. They represent mutable global state and their client classes are difficult to test due to the setup required for configuring the appropriate state. This causes the respective classes to have a lower testability score; they are tested less compared to other similar classes and the unit tests covering them are of a lower quality (have more test smells, such as General Fixture or Assertion Roulette). Similar observations can be made for some types of static methods. While the usage of static methods that modify state causes a production class to be tested less, for the ones that are part of utility classes or that only operate on parameters this is not the case. Constant do not have a negative impact on testability, neither in the classes in which they are declared nor in the corresponding client classes. Finally, the classes with static initialization blocks also seem a bit more testable, albeit for them it was difficult to evaluate this quality aspect due to the small number of instances present.

The change- and defect-proneness aspects were studied together because the procedures for assessing them are quite similar. The major difference is that for the latter only the commits which were categorized as bug-fixes are considered. Mutable global state instances, namely stateful singletons and static non-final attributes, are also very detrimental to change-proneness (same as for testability). The classes that have such instances were modified more frequently during commits and the number of changes that occurred is higher than for the rest of the production classes. Besides the ones that solely operate on parameters, the other static methods appear to have a negative impact on change-proneness; however, the effect is not as evident as for the mutable global state instances. Classes with static initialization blocks are in a similar situation. Finally, constants do not make the classes of which they are part of more change-prone; the average number of commits in which they were changed is comparable to that of similar classes.

The most important observation in terms of defect-proneness is that the classes that contain / utilize certain types of static constructs are less error-prone than they are change-prone. For example, the ratio between the average number of bug-fix commits in which stateful singletons / similar classes were modified is lower than the corresponding measurement when all the commits are taken into account. This observation also holds true for classes that have static non-final attributes and static methods that modify state. For the other types of static methods, constants, and static initialization blocks the values are very close to the ones obtained for change-proneness. All of the above suggest that using certain types of static constructs does have a negative effect on the software quality aspects investigated; nonetheless, there are static constructs (such as constants or static methods that solely operate on parameters) that do not affect these aspects.

7.3. Reflection

No major issues were encountered while implementing the proposed approach. The detection strategies for the different types of static constructs were successfully defined by leveraging the metrics already computed by Patrools and adding the ones that were missing (e.g., class has only private constructors). For testability we were able to obtain 1) coverage information by using JaCoCo through Maven / Gradle plugins and 2) data related to test smells with TSDetect. The scripts necessary for running these tools were easily integrated into DFAnalyser. The correlation between specific parts of the production code and the corresponding unit tests was also established using Patrools. In order to assess change-proneness we managed to extract fine-grained source code changes for the production classes using ChangeDistiller. For defect-proneness we categorized commits as bug-fixes based on information extracted both from the commit message and from the associated Jira issue tracker. In general, the proposed procedures were straightforward and easy to implement. Nonetheless, we consider that the ones related to the testability score and the bug-fix categorization could be improved, as will be explained in the future work section that follows.

Just like during implementation, the empirical study was conducted without any problems. The systems were selected based on a set of well-established criteria; we tried to choose systems that are different in terms of size and complexity, development practices, and testing effort. We did not encounter any issues while retrieving them from the corresponding Git repositories or when accessing the associated issue trackers. However, both while developing the approach and when conducting the study, we needed to make some decisions on how to proceed. Every time this had to be done, we provided the reasoning behind the decisions that were taken. As for any empirical study, there are ways in which it can be extended; we will analyse them in the next section.

7.4. Future work

Even though we tried to reach closure, there are several ways in which the proposed approach and the conducted study could be improved. We will discuss them, in no particular order, in the current section. Although the list is not exhaustive, these are the directions on which we will be focusing in the foreseeable future.

1. **Extending the empirical study:** as discussed in Section 6.2, there are several limitations to our study. We plan to address them by enhancing it in the following ways:

- *Additional Java systems*: first of all, we want to add more Java projects to our analyses. Although we studied a considerable number of systems which were selected based on a set of well-established criteria, there may still be some particular projects worth including. An important limitation of the study is that all the analysed systems are open-source. We hope that in the future we will have access to commercial projects. They might differ from open-source ones in terms of: 1) amount and types of design flaws present; 2) testing effort and quality of unit tests; 3) development practices. Studying such systems would ensure that our results are generalizable to any software project.
- *Other development technologies and programming paradigms:* another limitation is that all the analysed projects are implemented in Java. We are already pondering the possibility of reimplementing the tool in order to support other object-oriented programming languages (namely, C# and C++). The proposed approach should still apply, but the coding might need to be done in a C-family language. It will be interesting to see if the observed patterns are still valid for this type of systems.
  In addition to the development technology, we also want to study projects created by following other programming paradigms. As discussed in the previous chapter, static constructs are used very differently in embedded systems. This is why we are keen to extend the study to both imperative and declarative paradigms, including: procedural, functional, and logic programming. We will focus on the way in which static constructs are utilized, but will also examine their impact on the 3 quality aspects of interest.
- *Different development methodologies:* the methodologies play an important role in how a system is created. Different development practices have been observed for the chosen projects; however, none of the systems were created through Test-Driven Development. For TDD the test cases have to be written before the production code is implemented, thus the method through which we quantify testability might need to be adjusted accordingly. The latest agile development methodologies, such as extreme programming or lean development, will also be investigated. Doing this will add more credibility to the obtained results, thereby improving the quality of the study.

2. **Studying other design flaws:**

- *God Class:* this design flaw appears when a class "does too many things and knows too much" [99]. More specifically, this kind of instances perform most of the computations, delegate only minor responsibilities to a limited number of trivial classes, and utilize data from many other classes. The tool can be easily extended to detect God Classes because it can already

calculate the required metrics: 1) WMC (Weighted Method Count) to determine if a system's intelligence is distributed horizontally and as uniformly as possible or concentrated in several large classes [100]; 2) TCC (Tight Class Cohesion) to find classes with low communicative behaviour; 3) ATFD (Access to Foreign Data) to identify the classes that directly access data from other classes. After computing the 3 metrics, we can compare the values to the thresholds proposed in [101] and combine them into a detection strategy for God Class instances.
Due to their nature, we expect the God Classes to have a negative impact on software testability. Considering the large size of this type of classes, they are probably tested less compared to the rest of the production code. In the same vein, having too many responsibilities may cause smells to appear in their corresponding unit tests, thereby reducing the quality of the test code. The aforementioned problems could also affect the change- and defect-proneness of these classes. For example, more changes might be performed on God Classes during bug-fix commits compared to other similar classes.

- *Feature Envy:* this is another design flaw that could have an impact on the software quality aspects investigated. It appears when a method from a production class "accesses the data of another object more than its own data" [99]. Similar to the previous design flaw, the detection strategy is based on 3 metrics, namely: 1) ATFD (Access to Foreign Data); 2) LAA (Locality of Attribute Accesses) to determine if the method utilizes more attributes from other classes than its own attributes; 3) FDP (Foreign Data Providers) to calculate the number of classes the attributes belong to. The threshold values and the way in which the metrics are combined are thoroughly explained in [101]. Because they rely on data from many other classes the set-up required for testing might cause them to be covered less or with unit tests of a lower quality (e.g., with lots of test smells). Classes with Feature Envious methods may also change more often due to the fact that they depend on numerous other classes.
- *Object instantiations in constructors / methods:* instantiating objects instead of using Dependency Injection is a very common design problem. The issues that arise when doing this in constructors are discussed by Hevery in [17]; the most notable ones are: the violation of the Single Responsibility Principle, the difficulty in directly testing such constructors, and the fact that they cannot be subclasses or overridden for testing purposes. Most of the problems also appear when instantiating objects in production methods. We can extend the tool so that it can detect *new* statements within constructors / methods. For this flaw we will mainly focus on the testability aspect as it is less likely to have an impact on change- / defect-proneness.
- *Law of Demeter violations:* they occur when objects are received as parameters but never used directly; instead, their methods are called just to gain access to other objects. This design flaw should be detrimental to testability because multiple objects need to be configured in order to set-up the state properly. A class that contains such violations might also be change-prone since there are a lot of other classes on which it depends (that could be modified). The tool can easily detect Law of Demeter violations by querying the method call chain.

3. **Improving the way in which we compute the testability score:** although we evaluate both the quantity and the quality of the corresponding

unit tests when assessing the testability of a specific part of the production code, we still consider that the process through which we obtain the testability score could be refined. Especially from a quantitative perspective, more metrics could be included in addition to line coverage and the amount of unit tests for each production method. The evaluation on unit test quality could also be improved by including more test smells in the analysis.

4. **Refining the process through which we identify bug-fix commits:** we also want to perfect the method for categorizing commits as bug-fixes. Even though we leverage information extracted both from the commit message and from the corresponding issue tracker, there are still a lot of data available that can be utilized to improve this process. For example, the tool proposed in [102] could aid us in gathering additional information for refining the assessment. Furthermore, the commit history might also provide valuable data in this regard.
5. **Analysing everything at a lower level of granularity:** at the moment, most of the analyses are performed at class level. For example, a class is categorized as a singleton client if it utilizes at least one of its methods. We would like to make the analysis more fine-grained, therefore we need to be able to pinpoint which singleton methods are used by each of the methods from the respective class. This also applies to other static constructs, such as utility classes or mutable static attributes. Analysing everything at a lower level of granularity would also be beneficial for the models through which we evaluate the software quality aspects. Studying these aspects at method level would allow for a much more precise assessment. For testability we could determine which production methods are covered by a particular unit test; this would be interesting considering that some of the tests have significantly more smells than others. The method by which we categorize a commit as a bug-fix might also benefit from this refinement; for example, we would be able to search for method names (instead of class names) in the commit message and trace them back to their corresponding classes. The benefits of a more fine-grained analysis were highlighted for the source code changes that were extracted; being able to determine exactly what was changed during a commit was very important when assessing change-proneness.
6. **Proposing repair techniques for both production and test code:** the last research direction that we are considering is improving the code by refactoring the parts in which the problematic static constructs are present or by rewriting the unit tests so that the smells do not appear anymore. As an example, mutable global state instances could be eliminated by replacing the static non-final attributes with immutable ones while preserving the functionality. In the same vein, we could remove the General Fixture test smell by distributing the set-up logic to the appropriate unit tests; for example, only the tests that address a singleton client will configure the required singleton state, it will not be done in the set-up method of the test class. By performing these refactorings the quality of both the production and the test code will greatly improve.

In summary, this chapter discusses:

1. The contributions brought through our work:
    - the methodology for studying the evolution and the impact on software quality of any design flaw;
    - the model for quantifying class testability;
    - the process for identifying bug-fix commits and determining the fine-grained source code changes that occur between certain commits
    - a tool for investigating the aspects of interest;
    - an empirical study that answers the research questions for different types of static constructs;
    - a better understanding of static constructs, their evolution and the effect they have on 3 software quality aspects.
2. The main findings with regard to each research question:
    - that static constructs are heavily present in the code and are frequently utilized by other production classes;
    - that they are used less once a project reaches maturity compared to the earlier stages of its development;
    - that certain types of static constructs, such as mutable global state instances (stateful singletons and static non-final attributes) or static methods that modify their class's state, have a negative impact on the 3 quality aspects investigated.
3. What was accomplished thus far and what could have been done better.
4. Future work directions that we are currently considering:
    - improving the empirical study by adding more systems to it, including commercial ones and projects written in other languages / by following different programming paradigms;
    - investigating other design flaws such as God Class, Feature Envy, object instantiations in constructors / methods, or Law of Demeter violations;
    - refining the models through which we quantify the 3 quality aspects (e.g., adding more metrics to the testability score);
    - studying everything at a lower level of granularity;
    - suggesting repair techniques for the problematic parts of both production and test code.

References

[1] Brooks Jr, Frederick P. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E. Pearson Education India, 1995.

[2] Sommerville, Ian. "Software Engineering. International computer science series." *ed: Addison Wesley* (2004).

[3] Olan, Michael. "Unit testing: test early, test often." Journal of Computing Sciences in Colleges 19.2 (2003): 319-328.

[4] International Organization for Standardization. (2011). Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models (ISO Standard No. 25010:2011). https://www.iso.org/standard/35733.html.

[5] IEEE Standards Coordinating Committee. "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos." CA: IEEE Computer Society 169 (1990).

[6] R. Bache and M. Mullerburg, "Measures of testability as a basis for quality assurance," Softw. Eng. J., vol. 5, no. 2, pp. 86–92, 1990.

[7] Bruntink, Magiel, and Arie Van Deursen. "Predicting class testability using object-oriented metrics." *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE, 2004.

[8] Zhou, YuMing, et al. "An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems." *Science china information sciences* 55.12 (2012): 2800-2815.

[9] Bieman, James M., Anneliese Amschler Andrews, and Helen J. Yang. "Understanding change-proneness in OO software through visualization." *11th IEEE International Workshop on Program Comprehension, 2003.*. IEEE, 2003.

[10] Arvanitou, Elvira-Maria, et al. "A method for assessing class change proneness." *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. 2017.

[11] Mo, Ran, et al. "Architecture anti-patterns: Automatically detectable violations of design principles." *IEEE Transactions on Software Engineering* (2019).

[12] Shen, Vincent Yun, et al. "Identifying error-prone software—an empirical study." *IEEE Transactions on Software Engineering* 4 (1985): 317-324.

[13] Gyimóthy, Tibor, Rudolf Ferenc, and Istvan Siket. "Empirical validation of object-oriented metrics on open source software for fault prediction." *IEEE Transactions on Software engineering* 31.10 (2005): 897-910.

[14] Subramanyam, Ramanath, and Mayuram S. Krishnan. "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects." *IEEE Transactions on software engineering* 29.4 (2003): 297-310.

[15] Zhou, Yuming, and Hareton Leung. "Empirical analysis of object-oriented design metrics for predicting high and low severity faults." *IEEE Transactions on software engineering* 32.10 (2006): 771-789.

[16] Shatnawi, Raed, and Wei Li. "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process." *Journal of systems and software* 81.11 (2008): 1868-1882.

[17] Wolter, Jonathan, Russ Ruffer, and Miško Hevery. "Guide: Writing testable code." (2009): 1-38.

[18] Khomh, Foutse, Massimiliano Di Penta, and Yann-Gael Gueheneuc. "An exploratory study of the impact of code smells on software change-proneness." *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009.

[19] Kaur, Kamaldeep, and Shilpi Jain. "Evaluation of machine learning approaches for change-proneness prediction using code smells." *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications*. Springer, Singapore, 2017.

[20] Zhang, Xiaofang, Yida Zhou, and Can Zhu. "An empirical study of the impact of bad designs on defect proneness." *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2017.

[21] Alkhaeir, Tarek, and Bartosz Walter. "The effect of code smells on the relationship between design patterns and defects." *IEEE Access* 9 (2020): 3360-3373.

[22] D'Ambros, Marco, Alberto Bacchelli, and Michele Lanza. "On the impact of design flaws on software defects." *2010 10th International Conference on Quality Software*. IEEE, 2010.

[23] Eken, Beyza, et al. "An empirical study on the effect of community smells on bug prediction." *Software Quality Journal* 29.1 (2021): 159-194.

[24] Khalid, Sadaf, Saima Zehra, and Fahim Arif. ″Analysis of object oriented complexity and testability using object oriented design metrics.″ Proceedings of the 2010 National Software Engineering Conference. 2010.

[25] Chen, Jie-Cherng, and Sun-Jen Huang. ″An empirical analysis of the impact of software development problem factors on software maintainability.″ Journal of Systems and Software 82.6 (2009): 981-992.

[26] Soni, Devpriya. "Evaluation of Understandability of Object-Oriented Design." *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*. Springer, Berlin, Heidelberg, 2013.

[27] Arce, Iván, et al. "Avoiding the top 10 software security design flaws." *IEEE Computer Society Center for Secure Design (CSD), Tech. Rep* (2014).

[28] Why Static Code is Bad, https://objcsharp.wordpress.com/2013/07/08/why-static-code-is-bad/

[29] Marsavina, Cosmin. "Studying the Evolution of Static Methods and their Effect on Class Testability." *2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 2020.

[30] Marsavina, Cosmin. "Understanding the Impact of Mutable Global State on the Defect Proneness of Object-Oriented Systems." *2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 2020.

[31] Brown, William H., et al. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[32] Marinescu, Radu. "Detecting design flaws via metrics in object-oriented systems." *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*. IEEE, 2001.

[33] Marinescu, Radu. "Detection strategies: Metrics-based rules for detecting design flaws." Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. IEEE, 2004.

[34] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2015). Mining version histories for detecting code smells.

[35] Kessentini, Marouane, et al. "Design defects detection and correction by example." Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. IEEE, 2011.

[36] Moha, Naouel, et al. "DECOR: A method for the specification and detection of code and design smells." Software Engineering, IEEE Transactions on 36.1 (2010): 20-36.

[37] Wegrzynowicz, Patrycja, and Krzysztof Stencel. "Towards a comprehensive test suite for detectors of design patterns." *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009.

[38] Van Rompaey, Bart, et al. "On the detection of test smells: A metrics-based approach for general fixture and eager test." *IEEE Transactions on Software Engineering* 33.12 (2007): 800-817.

[39] Bavota, Gabriele, et al. "An empirical analysis of the distribution of unit test smells and their impact on software maintenance." *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012.

[40] Bavota, Gabriele, et al. "Are test smells really harmful? An empirical study." (2014).

[41] Jianping, Fu, Liu Bin, and Lu Minyan. "Present and future of software testability analysis." *Computer Application and System Modeling (ICCASM), 2010 International Conference on*. Vol. 15. IEEE, 2010.

[42] Tahir, Amjed. *A Study on Software Testability and the Quality of Testing In Object-Oriented Systems*. Diss. University of Otago, 2016.

[43] D. Rapu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection." *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE, 2004.

[44] Marsavina, Cosmin, Daniele Romano, and Andy Zaidman. "Studying fine-grained co-evolution patterns of production and test code." *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014.

[45] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining." *2012 16th European conference on software maintenance and reengineering*. IEEE, 2012.

[46] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey." *Proceedings of the 4th Workshop on Refactoring Tools*. 2011.

[47] S. Vaucher, F. Khomh, N. Moha, and Y. G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes." *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009.

[48] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems." *2009 3rd international symposium on empirical software engineering and measurement*. IEEE, 2009.

[49] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad (and whether the smells go away).” *IEEE Transactions on Software Engineering* 43.11 (2017): 1063-1088.

[50] A. Chatzigeorgiou and A. Manakos, “Investigating the evolution of bad smells in object-oriented code.” *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010.

[51] M. Martinez and M. Monperrus, “Coming: A tool for mining change pattern instances from git commits.” *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019.

[52] Mouchawrab, Samar, Lionel C. Briand, and Yvan Labiche. "A measurement framework for object-oriented software testability." Information and software technology 47.15 (2005): 979-997.

[53] Bruntink, Magiel, and Arie van Deursen. "An empirical study into class testability." Journal of systems and software 79.9 (2006): 1219-1232.

[54] Zhou, YuMing, et al. "An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems." Science china information sciences 55.12 (2012): 2800-2815.

[55] Nikfard, Pourya, et al. "An Empirical Analysis of a Testability Model."Informatics and Creative Multimedia (ICICM), 2013 International Conference on. IEEE, 2013.

[56] Han, Ah-Rim, et al. "Behavioral dependency measurement for change-proneness prediction in UML 2.0 design models." *2008 32nd Annual IEEE International Computer Software and Applications Conference*. IEEE, 2008.

[57] Han, Ah-Rim, et al. "Measuring behavioral dependency for improving change-proneness prediction in UML-based design models." *Journal of Systems and Software* 83.2 (2010): 222-234.

[58] Kumar, Lov, Santanu Kumar Rath, and Ashish Sureka. "Empirical analysis on effectiveness of source code metrics for predicting change-proneness." *Proceedings of the 10th Innovations in Software Engineering Conference*. 2017.

[59] Agrawal, Anushree, and Rakesh Kumar Singh. "Empirical validation of OO metrics and machine learning algorithms for software change proneness prediction." *Towards Extensible and Adaptable Methods in Computing*. Springer, Singapore, 2018. 69-84.

[60] Posnett, Daryl, Christian Bird, and Prem Dévanbu. "An empirical study on the influence of pattern roles on change-proneness." *Empirical Software Engineering* 16.3 (2011): 396-423.

[61] Koru, A. Gunes, Dongsong Zhang, and Hongfang Liu. "Modeling the effect of size on defect proneness for open-source software." *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 2007.

[62] Olague, Hector M., et al. "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes." *IEEE Transactions on software Engineering* 33.6 (2007): 402-419.

[63] Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. ”A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction.” Proceedings of the 30th international conference on Software engineering. 2008.

[64] Gray, David, et al. "Software defect prediction using static code metrics underestimates defect-proneness." *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2010.

[65] Soni, Devpriya. "Evaluation of Understandability of Object-Oriented Design." *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*. Springer, Berlin, Heidelberg, 2013.

[66] Southam, Blaine R. "Guide-Writing Testable Code." (2009).

[67] Hevery, Misko. "Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process."*Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2008.

[68] Sabane, Aminata, et al. "A study on the relation between antipatterns and the cost of class unit testing." *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013.

[69] Baudry, Benoit, et al. "Measuring and improving design patterns testability."*Software Metrics Symposium, 2003. Proceedings. Ninth International*. IEEE, 2003.

[70] Tahir, A., S.G. MacDonell, and J. Buchan, Understanding class-level testability through dynamic analysis, in International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). 2014: Lisbon, Portugal. p. 38-47.

[71] S. Jungmayr, "Testability measurement and software dependencies," 2002.

[72] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects." *2010 10th International Conference on Quality Software*. IEEE, 2010.

[73] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems." *IBM Journal of Research and Development* 56.5 (2012): 9-1.

[74] F. Khomh, M. Di Penta, Y. G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness." *Empirical Software Engineering* 17.3 (2012): 243-275.

[75] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation." *Empirical Software Engineering* 23.3 (2018): 1188-1221.

[76] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection?–An empirical study." *Information and Software Technology* 55.12 (2013): 2223-2242.

[77] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort." *IEEE Transactions on Software Engineering* 39.8 (2012): 1144-1156.

[78] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23.4 (2014): 1-39.

[79] Badri, Mourad, and Fadel Toure. "Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes." *Journal of Software Engineering and Applications* 5.7 (2012): 513.

[80] Static Methods are Death to Testability, http://misko.hevery.com/2008/12/15/static-methods-are-death-to-testability/

[81] Why Static Code is Bad, https://objcsharp.wordpress.com/2013/07/08/why-static-code-is-bad/

[82] How Not To Kill Your Testability Using Statics, http://kunststube.net/static/

[83] Static Methods Will Shock You, https://simpleprogrammer.com/2010/01/29/static-methods-will-shock-you/

[84] Feathers, Michael. "Testable Java."

[85] http://www.drdobbs.com/javas-new-considered-harmful/184405016

[86] http://misko.hevery.com/2008/07/08/how-to-think-about-the-new-operator/

[87] Feathers, Michael. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[88] Gil, Joseph Yossi, and Itay Maman. "Micro patterns in Java code." *ACM SIGPLAN Notices* 40.10 (2005): 97-116.

[89] Geary, David. "Simply Singleton, Navigate the deceptively simple Singleton pattern." JavaWorld How-To-Java, April (2003).

[90] Hevery, Misko. "Singletons are pathological liara." (2008).

[91] K. Stencel and P. Węgrzynowicz, "Implementation variants of the singleton design pattern." *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems*." Springer, Berlin, Heidelberg, 2008.

[92] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, "Eclemma-jacoco java code coverage library." (2011).

[93] Spadini, Davide, et al. "On the relation of test smells to software code quality." *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018.

[94] A. Peruma, et al. "tsDetect: an open source test smells detection tool." *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020.

[95] P. F. Mihancea, "Patrools: Visualizing the Polymorphic Usage of Class Hierarchies." *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 2010.

[96] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller." *IEEE software* 26.1 (2009): 26-33.

[97] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in Symposium on the Foundations of Software Engineering (FSE). ACM, 2012, p. 33.

[98] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. Empirical studies of software engineering : A roadmap. The future of Software engineering, pages 345–355, 2000.

[99] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st ed. John Wiley and Sons, March 1998.

[100] Sousa, Bruno & Bigonha, Mariza & Ferreira, Kecia. (2017). Evaluating Co-Occurrence of GOF Design Patterns with God Class and Long Method Bad Smells. 10.5753/sbsi.2017.6068.

[101] Gradišnik, Mitja, et al. "Adapting God Class thresholds for software defect prediction: A case study." *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2019.

[102] Bachmann, Adrian, et al. "The missing links: bugs and bug-fix commits." *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 2010.

Publications

[A1]. **Marsavina, Cosmin**, Daniele Romano, and Andy Zaidman. "Studying fine-grained co-evolution patterns of production and test code." *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014.

Cited by:

- Beller, Moritz, et al. "When, how, and why developers (do not) test in their IDEs." *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015.
- Labuschagne, Adriaan, Laura Inozemtseva, and Reid Holmes. "Measuring the cost of regression testing in practice: A study of Java projects using continuous integration." *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017.
- Beller, Moritz, et al. "Developer testing in the ide: Patterns, beliefs, and behavior." *IEEE Transactions on Software Engineering* 45.3 (2017): 261-284.
- Macho, Christian, Shane McIntosh, and Martin Pinzger. "Automatically repairing dependency-related build breakage." *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018.
- Grano, Giovanni, Fabio Palomba, and Harald C. Gall. "Lightweight assessment of test-case effectiveness using source-code-quality indicators." *IEEE Transactions on Software Engineering* (2019).
- Rodríguez-Pérez, Gema, et al. "How bugs are born: a model to identify how bugs are introduced in software components." *Empirical Software Engineering* 25.2 (2020): 1294-1340.
- Imtiaz, Javaria, et al. "A systematic literature review of test breakage prevention and repair techniques." *Information and Software Technology* 113 (2019): 1-19.
- Soetens, Quinten David, Romain Robbes, and Serge Demeyer. "Changes as first-class citizens: A research perspective on modern software tooling." *ACM Computing Surveys (CSUR)* 50.2 (2017): 1-38.
- Levin, Stanislav, and Amiram Yehudai. "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes." *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017.
- Rapos, Eric J., and James R. Cordy. "Examining the co-evolution relationship between Simulink models and their test cases." *Proceedings of the 8th International Workshop on Modeling in Software Engineering*. 2016.
- Vidács, László, and Martin Pinzger. "Co-evolution analysis of production and test code by learning association rules of changes." *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2018.

- Chahal, Kuljit Kaur, and Munish Saini. "Open source software evolution: a systematic literature review (Part 1)." *International Journal of Open Source Software and Processes (IJOSSP)* 7.1 (2016): 1-27.
- Alsolami, Nada, Qasem Obeidat, and Mamdouh Alenezi. "Empirical analysis of object-oriented software test suite evolution." *International Journal of Advanced Computer Science and Applications* 10.11 (2019).
- Rapolu, Swetha, and Tooraj Nikoubin. "Fast and energy efficient FinFET full adders with Cell Design Methodology (CDM)." *2015 6th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE, 2015.
- Zampetti, Fiorella, et al. "Demystifying the adoption of behavior-driven development in open source projects." *Information and Software Technology* 123 (2020): 106311.
- Inozemtseva, Laura Michelle McLean. "Data science for software maintenance." (2017).
- Soetens, Quinten David. *Change-Based Software Engineering: Using Reified Changes for Test Selection and Refactoring Reconstruction*. Diss. Universiteit Antwerpen (Belgium), 2015.
- Spadini, Davide. "Practices and tools for better software testing." *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018.
- Labuschagn, Adriaan, Laural Inozemtseva, and Reid Holmes. "Measuring the cost of regression testing in practice." *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the foundations of Software Engineering. Retrieved from*. Vol. 10. No. 3106237.3106288. 2017.
- Macho, Christian, et al. "The nature of build changes." *Empirical Software Engineering* 26.3 (2021): 1-53.
- Labuschagne, Adriaan. *Continuous integration build failures in practice*. MS thesis. University of Waterloo, 2016.
- Wang, Peipei, et al. "Demystifying regular expression bugs." *Empirical Software Engineering* 27.1 (2022): 1-35.
- Le Dilavrec, Quentin, et al. "Untangling Spaghetti of Evolutions in Software Histories to Identify Code and Test Co-evolutions." *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021.
- Kim, Misoo, Youngkyoung Kim, and Eunseok Lee. "Denchmark: A Bug Benchmark of Deep Learning-related Software." *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021.
- Zaidman, Andy, M. Daniel, and Jesus M. Gonzalez-Barahona. "How bugs are born: a model to identify how bugs are introduced in software components." (2020).
- Kim, Misoo, and Eunseok Lee. "Are datasets for information retrieval-based bug localization techniques trustworthy?." *Empirical Software Engineering* 26.3 (2021): 1-66.

- Wang, Sinan, et al. "Understanding and Facilitating the Co-Evolution of Production and Test Code." *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021.
- Levin, Stanislav, and Amiram Yehudai. "Processing Large Datasets of Fined Grained Source Code Changes." *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.
- Saini, Munish, and Kuljit Kaur Chahal. "A Systematic Review of Attributes and Techniques for Open Source Software Evolution Analysis." *Research Anthology on Usage and Development of Open Source Software* (2021): 1-23.
- Rwemalika, Renaud. *On the Maintenance of System User Interactive Tests*. Diss. University of Luxembourg, Luxembourg, Luxembourg, 2021.
- Klammer, Claus, Georg Buchgeher, and Albin Kern. "A retrospective of production and test code co-evolution in an industrial project." *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. IEEE, 2018.
- Rodríguez-Pérez, G., et al. "How bugs are born."
- Wang, Peipei, et al. "Demystifying Regular Expression Bugs: A comprehensive study on regular expression bug causes, fixes, and testing." *arXiv preprint arXiv:2104.09693* (2021).
- Rapos, Eric J. *Supporting simulink model management*. Diss. Queen's University (Canada), 2017.
- Wang, Peipei. *Analyses of Regular Expression Usage in Software Development*. Diss. North Carolina State University, 2021.
- Harbi, Fahad. *Exploring the complexity of risk interaction with success factors and success criteria in software projects*. The University of Liverpool (United Kingdom), 2017.
- Kazerouni, Ayaan Mehdi. *Measuring the Software Development Process to Enable Formative Feedback*. Diss. Virginia Tech, 2020.
- Kim, Jungil, and Eunjoo Lee. "A Change Recommendation Approach Using Change Patterns of a Corresponding Test File." *Symmetry* 10.11 (2018): 534.
- Miranda, Charles, et al. "Uma Análise da Co-Evolução de Teste em Projetos de Software no GitHub." *Anais do IX Workshop de Visualização, Evolução e Manutenção de Software*. SBC, 2021.

[A2]. **Marsavina, Cosmin**. "Understanding the Impact of Mutable Global State on the Defect Proneness of Object-Oriented Systems." *2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 2020.

[A3]. **Marsavina, Cosmin**. "Studying the Evolution of Static Methods and their Effect on Class Testability." *2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 2020.
[A4]. Șerban, D. A., **Marșavina, C.**, Coșa, A. V., Belgiu, G., & Negru, R. (2021). A Study of Yielding and Plasticity of Rapid Prototyped ABS. *Mathematics*, *9*(13), 1495.